Operators

Operators in DocETL are designed for semantically processing unstructured data. They form the building blocks of data processing pipelines, allowing you to transform, analyze, and manipulate datasets efficiently.

Overview

- Datasets contain items, where a item is an object in the JSON list, with fields and values. An item here could be simple text chunk or a document reference.
- DocETL provides several operators, each tailored for specific unstructured data processing tasks.
- By default, operations are parallelized on your data using multithreading for improved performance.

> **Caching in DocETL**
>
> DocETL employs caching for all LLM calls and partially-optimized plans. The cache is stored in the `.cache/docetl/general` and `.cache/docetl/llm` directories within your home directory. This caching mechanism helps to improve performance and reduce redundant API calls when running similar operations or reprocessing data.

Common Attributes

All operators share some common attributes:

- `name`: A unique identifier for the operator.
- `type`: Specifies the type of operation (e.g., "map", "reduce", "filter").

LLM-based operators have additional attributes:

- `prompt`: A Jinja2 template that defines the instruction for the language model.
- `output`: Specifies the schema for the output from the LLM call.
- `model` (optional): Allows specifying a different model from the pipeline default.
- `litellm_completion_kwargs` (optional): Additional parameters to pass to LiteLLM completion calls.

DocETL uses LiteLLM to execute all LLM calls, providing support for 100+ LLM providers including OpenAI, Anthropic, Azure, and more. You can pass any LiteLLM completion arguments using the `litellm_completion_kwargs` field.

Example:

```
- name: extract_insights
  type: map
  model: gpt-4o-mini
  litellm_completion_kwargs:
    max_tokens: 500          # limit response length
    temperature: 0.7         # control randomness
    top_p: 0.9              # nucleus sampling parameter
  prompt: |
    Analyze the following user interaction log:
    {{ input.log }}

    Extract 2-3 main insights from this log, each being 1-2 words, to help
inform future product development. Consider any difficulties or pain points
the user may have had. Also provide 1-2 supporting actions for each insight.
    Return the results as a list of dictionaries, each containing 'insight'
and 'supporting_actions' keys.
  output:
    schema:
      insights: "list[{insight: string, supporting_actions: list[string]}]"
```

Input and Output

Prompts can reference any fields in the data, including:

- Original fields from the input data.
- Fields synthesized by previous operations in the pipeline.

For map operations, you can only reference `input`, but in reduce operations, you can reference `inputs` (since it's a list of inputs).

Example:

```
prompt: |
  Summarize the user behavior insights for the country: {{ inputs[0].country
}}

  Insights and supporting actions:
  {% for item in inputs %}
  - Insight: {{ item.insight }}
  Supporting actions:
  {% for action in item.supporting_actions %}
  - {{ action }}
  {% endfor %}
  {% endfor %}
```

What happens if the input is too long?

When the input data exceeds the token limit of the LLM, DocETL automatically truncates tokens from the middle of the data to make it fit in the prompt. This approach preserves the beginning and end of the input, which often contain crucial context.

A warning is displayed whenever truncation occurs, alerting you to potential loss of information:

```
WARNING: Input exceeded token limit. Truncated 500 tokens from the middle of
the input.
```

If you frequently encounter this warning, consider using DocETL's optimizer or breaking down your input yourself into smaller chunks to handle large inputs more effectively.

Output Schema

The `output` attribute defines the structure of the LLM's response. It supports various data types (see schemas for more details):

- `string` (or `str`, `text`, `varchar`): For text data
- `integer` (or `int`): For whole numbers
- `number` (or `float`, `decimal`): For decimal numbers
- `boolean` (or `bool`): For true/false values
- `list`: For arrays or sequences of items
- objects: Using notation `{field: type}`
- `enum`: For a set of possible values

Example:

```yaml
output:
  schema:
    insights: "list[{insight: string, supporting_actions: string}]"
    detailed_summary: string
```

Keep Output Types Simple

It's recommended to keep output types as simple as possible. Complex nested structures may be difficult for the LLM to consistently produce, potentially leading to parsing errors. The structured output feature works best with straightforward schemas. If you need complex data structures, consider breaking them down into multiple simpler operations.

For example, instead of:

```
output:
  schema:
    insights: "list[{insight: string, supporting_actions: list[{action: string,
priority: integer}]}]"
```

Consider:

```
output:
  schema:
    insights: "list[{insight: string, supporting_actions: string}]"
```

And then use a separate operation to further process the supporting actions if needed.

Read more about schemas in the schemas section.

Validation

Validation is a first-class citizen in DocETL, ensuring the quality and correctness of processed data.

Basic Validation

LLM-based operators can include a `validate` field, which accepts a list of Python statements:

```
validate:
  - len(output["insights"]) >= 2
  - all(len(insight["supporting_actions"]) >= 1 for insight in
output["insights"])
```

Access variables using dictionary syntax: `output["field"]`. Note that you can't access `input` docs in validation, but the output docs should have all the fields from the input docs (for non-reduce operations), since fields pass through unchanged.

The `num_retries_on_validate_failure` attribute specifies how many times to retry the LLM if any validation statements fail.

Advanced Validation: Gleaning

Gleaning is an advanced validation technique that uses LLM-based validators to refine outputs iteratively.

To enable gleaning, specify:

- `validation_prompt`: Instructions for the LLM to evaluate and improve the output.
- `num_rounds`: The maximum number of refinement iterations.
- `model` (optional): The model to use for the LLM executing the validation prompt. Defaults to the model specified for this operation. **Note that if the validator LLM determines the output needs to be improved, the final output will be generated by the model specified for this operation.**
- `if` (optional): A Python boolean expression (evaluated with `safe_eval`) that refers to **fields in the current `output`**. If the expression evaluates to `False`, DocETL skips gleaning entirely.

Example:

```
gleaning:
  num_rounds: 1
  validation_prompt: |
    Evaluate the extraction for completeness and relevance:
    1. Are all key user behaviors and pain points from the log addressed in
the insights?
    2. Are the supporting actions practical and relevant to the insights?
    3. Is there any important information missing or any irrelevant
information included?
```

This approach allows for *context-aware* validation and refinement of LLM outputs. Note that it is expensive, since it at least doubles the number of LLM calls required for each operator.

Example map operation (with a different model for the validation prompt):

```
- name: extract_insights
  type: map
  model: gpt-4o
  prompt: |
    From the user log below, list 2-3 concise insights (1-2 words each) and 1-
2 supporting actions per insight.
    Return as a list of dictionaries with 'insight' and 'supporting_actions'.
    Log: {{ input.log }}
  output:
    schema:
      insights_summary: "string"
  gleaning:
    if: "len(output['insights_summary']) < 10"  # Only refine if summary is
too short
```

```
    num_rounds: 2 # Will refine up to 2 times if needed
    model: gpt-4o-mini
    validation_prompt: |
      There should be at least 2 insights, and each insight should have at
least 1 supporting action.
```

Choosing a Different Model for Validation

You may want to use a different model for the validation prompt. For example, you can use a more powerful (and expensive) model for generating outputs, but a cheaper model for validation—especially if the validation only checks a single aspect. This approach helps reduce costs while still ensuring quality, since the final output is always produced by the more capable model.

Conditional Gleaning

You can also use the `if` field to conditionally skip gleaning. For example, if you only want to glean if the output is too short, you can use:

```
gleaning:
  if: "len(output['insights_summary']) < 10"
  num_rounds: 2
```

If the `if` field evaluates to `False`, DocETL skips gleaning entirely. Or, if the `if` field does not exist, DocETL will always glean.

How Gleaning Works

Gleaning is an iterative process that refines LLM outputs using context-aware validation. Here's how it works:

1. **Initial Operation**: The LLM generates an initial output based on the original operation prompt.
2. **Validation**: The validation prompt is appended to the chat thread, along with the original operation prompt and output. This is submitted to the LLM. *Note that the validation prompt doesn't need any variables, since it's appended to the chat thread.*
3. **Assessment**: The LLM responds with an assessment of the output according to the validation prompt. The model used for this step is specified by the `model` field in the `gleaning` dictionary field, or defaults to the model specified for that operation.
4. **Decision**: The system interprets the assessment:
    - If there's no error or room for improvement, the current output is returned.

- If improvements are suggested, the process continues.

5. **Refinement**: If improvements are needed:
   - A new prompt is created, including the original operation prompt, the original output, and the validator feedback.
   - This is submitted to the LLM to generate an improved output.
6. **Iteration**: Steps 2-5 are repeated until either:
   - The validator has no more feedback (i.e., the evaluation passes), or
   - The number of iterations exceeds `num_rounds`.
7. **Final Output**: The last refined output is returned.

Note that gleaning can significantly increase the number of LLM calls for each operator, potentially doubling it at minimum. While this increases cost and latency, it can lead to higher quality outputs for complex tasks.

Schemas

In DocETL, schemas play an important role in defining the structure of output from LLM operations. Every LLM call in DocETL is associated with an output schema, which specifies the expected format and types of the output data.

Overview

- Schemas define the structure and types of output data from LLM operations.
- They help ensure consistency and facilitate downstream processing.
- DocETL uses structured outputs or tool API to enforce these schemas.

> **Schema Simplicity**
>
> We've observed that **the more complex the output schema is, the worse the quality of the output tends to be**. Keep your schemas as simple as possible for better results.

Defining Schemas

Schemas are defined in the `output` section of an operator. They support various data types:

Type	Aliases	Description
`string`	`str`, `text`, `varchar`	For text data
`integer`	`int`	For whole numbers
`number`	`float`, `decimal`	For decimal numbers
`boolean`	`bool`	For true/false values
`enum`	-	For a set of possible values

Type	Aliases	Description
`list`	-	For arrays or sequences of items (must specify element type)
Objects	-	Using notation `{field: type}`

Filter Operation Schemas

Filter operation schemas must have a boolean type output field. This is used to determine whether each item should be included or excluded based on the filter criteria.

Examples

Simple Schema

```yaml
output:
  schema:
    summary: string
    sentiment: string
    include_item: boolean # For filter operations
```

Complex Schema

```yaml
output:
  schema:
    insights: "list[{insight: string, confidence: number}]"
    metadata: "{timestamp: string, source: string}"
```

Lists and Objects

Lists in schemas must specify their element type:

- `list[string]` : A list of strings
- `list[int]` : A list of integers
- `list[{name: string, age: integer}]` : A list of objects

Objects are defined using curly braces and must have typed fields:

- `{name: string, age: integer, is_active: boolean}`

Complex List Example

```
output:
  schema:
    users: "list[{name: string, age: integer, hobbies: list[string]}]"
```

Make sure that you put the type in quotation marks, if it references an object type (i.e., has curly braces)! Otherwise the yaml won't compile!

Enum Types

You can also specify enum types, which will be validated against a set of possible values. Suppose we have an operation to extract sentiments from a document, and we want to ensure that the sentiment is one of the three possible values. Our schema would look like this:

```
output:
  schema:
    sentiment: "enum[positive, negative, neutral]"
```

You can also specify a list of enum types (say, if we wanted to extract *multiple* sentiments from a document):

```
output:
  schema:
    possible_sentiments: "list[enum[positive, negative, neutral]]"
```

How We Enforce Schemas

DocETL uses structured outputs or tool API to enforce schema typing. This ensures that the LLM outputs adhere to the specified schema, making the results more consistent and easier to process in subsequent operations.

DocETL supports two output modes that determine how the LLM generates structured outputs:

Tools Mode (Default)

Uses the OpenAI tools/function calling API to enforce schema structure. This is the default mode and provides robust schema validation.

```
output:
  schema:
```

```
    summary: string
    sentiment: string
  mode: "tools"  # Optional - this is the default
```

Structured Output Mode

Uses LiteLLM's structured output feature with JSON schema validation. This mode can provide more reliable schema adherence for complex outputs.

```
output:
  schema:
    insights: "list[{insight: string, confidence: number}]"
  mode: "structured_output"
```

> **When to Use Structured Output Mode**
>
> Consider using `structured_output` mode when:
>
> - You have complex nested schemas with lists and objects
> - You need more consistent schema adherence
> - You're experiencing schema validation issues with tools mode
>
> The tools mode remains the default and works well for most use cases.

Mode Configuration

The output mode can be configured in the `output` section of any operation:

```
operations:
  - name: analyze_text
    type: map
    prompt: "Analyze the following text..."
    output:
      schema:
        topics: "list[{topic: string, relevance: number}]"
      mode: "structured_output"  # or "tools"
    model: gpt-4o-mini
```

If no mode is specified, DocETL defaults to `"tools"` mode for backward compatibility.

Best Practices

1. Keep output fields simple and use string types whenever possible.

2. Only use structured fields (like lists and objects) when necessary for downstream analysis or reduce operations.
3. If you need to reference structured fields in downstream operations, consider breaking complex structures into multiple simpler operations.

Schema Optimization

If you find your schema becoming too complex, consider breaking it down into multiple operations. This can improve both the quality of LLM outputs and the manageability of your pipeline.

Breaking Down Complex Schemas

Instead of:

```yaml
output:
  schema:
    summary: string
    key_points: "list[{point: string, sentiment: string}]"
```

Consider:

```yaml
output:
  schema:
    summary: string
    key_points: "string"
```

Where in the prompt you can say something like: `In your key points, please include the sentiment of each point.`

The only reason to use the complex schema is if you need to do an operation at the point level, like resolve them and reduce on them.

By following these guidelines and best practices, you can create effective schemas that enhance the performance and reliability of your DocETL operations.

Pipelines

Pipelines in DocETL are the core structures that define the flow of data processing. They orchestrate the application of operators to datasets, creating a seamless workflow for complex chunk processing tasks.

Components of a Pipeline

A pipeline in DocETL consists of five main components:

1. **Default Model**: The language model to use for the pipeline.
2. **System Prompts**: A description of your dataset and the "persona" you'd like the LLM to adopt when analyzing your data.
3. **Datasets**: The input data sources for your pipeline.
4. **Operators**: The processing steps that transform your data.
5. **Pipeline Specification**: The sequence of steps and the output configuration.

Default Model

You can set the default model for a pipeline in the YAML configuration file. If no model is specified at the operation level, the default model will be used.

You can also tell DocETL to skip the dataset-level cache for the entire pipeline by enabling `bypass_cache`. When set to `true`, DocETL will neither read from nor write to its cache for any operation in that pipeline—which is helpful when you want to force fresh executions during development or debugging.

```yaml
default_model: gpt-4o-mini
bypass_cache: true  # optional - defaults to false
```

`bypass_cache` can still be overridden at the operator level if required.

You can also specify default API base URLs for language models and embeddings if you're hosting your own models with an OpenAI-compatible API:

> **Note**
>
> If you're hosting your own models with an OpenAI-compatible API, you can specify the base URLs:
>
> ```
> default_lm_api_base: https://your-custom-llm-endpoint.com/v1
> default_embedding_api_base: https://your-custom-embedding-endpoint.com/v1
> ```
>
> This is particularly useful when working with self-hosted models or services like Ollama, LM Studio, or other API-compatible LLM servers.

System Prompts

System prompts provide context to the language model about the data it's processing and the role it should adopt. This helps guide the model's responses to be more relevant and domain-appropriate. There are two key components to system prompts in DocETL:

1. **Dataset Description**: A concise explanation of what kind of data the model will be processing.
2. **Persona**: The role or perspective the model should adopt when analyzing the data.

Here's an example of how to define system prompts in your pipeline configuration:

```
system_prompt:
  dataset_description: a collection of transcripts of doctor visits
  persona: a medical practitioner analyzing patient symptoms and reactions to
medications
```

Datasets

Datasets define the input data for your pipeline. They are collections of items/chunks, where each item/chunk is an object in a JSON list (or row in a CSV file). Datasets are typically specified in the YAML configuration file, indicating the type and path of the data source. For example:

```
datasets:
  user_logs:
    type: file
    path: "user_logs.json"
```

Dynamic Data Loading

DocETL supports dynamic data loading, allowing you to process various file types by specifying a key that points to a path or using a custom parsing function. This feature is

particularly useful for handling diverse data sources, such as audio files, PDFs, or any other non-standard format.

To implement dynamic data loading, you can use parsing tools in your dataset configuration. Here's an example:

```yaml
datasets:
  audio_transcripts:
    type: file
    source: local
    path: "audio_files/audio_paths.json"
    parsing_tools:
      - input_key: audio_path
        function: whisper_speech_to_text
        output_key: transcript
```

In this example, the dataset configuration specifies a JSON file (audio_paths.json) that contains paths to audio files. The parsing_tools section defines how to process these files:

- `input_key`: Specifies which key in the JSON contains the path to the audio file. In this example, each object in the dataset should have a "audio_path" key, that represents a path to an audio file or mp3.
- `function`: Names the parsing function to use (in this case, the built-in whisper_speech_to_text function for audio transcription).
- `output_key`: Defines the key where the processed data (transcript) will be stored. You can access this in the pipeline in any prompts with the `{{ input.transcipt }}` syntax.

This approach allows DocETL to dynamically load and process various file types, extending its capabilities beyond standard JSON or CSV inputs. You can use built-in parsing tools or define custom ones to handle specific file formats or data processing needs. See the Custom Parsing documentation for more details.

> **Note**
>
> Currently, DocETL only supports JSON files or CSV files as input datasets. If you're interested in support for other data types or cloud-based datasets, please reach out to us or join our open-source community and contribute! We welcome new ideas and contributions to expand the capabilities of DocETL.

Dataset Description and Persona

You can define a description of your dataset and persona you want the LLM to adopt when executing operations on your dataset. This is useful for providing context to the

LLM and for optimizing the operations.

```
system_prompt: # This is optional, but recommended for better performance. It
is applied to all operations in the pipeline.
  dataset_description: a collection of transcripts of doctor visits
  persona: a medical practitioner analyzing patient symptoms and reactions to
medications
```

Operators

Operators are the building blocks of your pipeline, defining the transformations and analyses to be performed on your data. They are detailed in the Operators documentation. Operators can include map, reduce, filter, and other types of operations.

Pipeline Specification

The pipeline specification outlines the sequence of steps to be executed and the final output configuration. It typically includes:

- Steps: The sequence of operations to be applied to the data.
- Output: The configuration for the final output of the pipeline.

For example:

```
pipeline:
  steps:
    - name: analyze_user_logs
      input: user_logs
      operations:
        - extract_insights
        - unnest_insights
        - summarize_by_country
  output:
    type: file
    path: "country_summaries.json"
    intermediate_dir: "intermediate_data" # Optional: If you want to store
intermediate outputs in a directory
```

For a practical example of how these components come together, refer to the Tutorial, which demonstrates a complete pipeline for analyzing user behavior data.

Optimization

Sometimes, finding the optimal pipeline for your task can be challenging. You might wonder:

Questions

- Will a single LLM call suffice for your task?
- Do you need to decompose your task or data further for better results?

To address these questions and improve your pipeline's performance, you can use DocETL to build an optimized version of your pipeline.

The DocETL Optimizer

The DocETL optimizer is designed to decompose operators (and sequences of operators) into their own subpipelines, potentially leading to higher accuracy.

Example

A map operation attempting to perform multiple tasks can be decomposed into separate map operations, ultimately improving overall accuracy. For example, consider a map operation on student survey responses that tries to:

1. Extract actionable suggestions for course improvement
2. Identify potential interdisciplinary connections

This could be optimized into two *separate* map operations:

- Suggestion Extraction: Focus solely on identifying concrete, actionable suggestions for improving the course.

```
prompt: |
  From this student survey response, extract any specific, actionable
suggestions
  for improving the course. If no suggestions are present, output 'No
suggestions found.':
  '{{ input.response }}'
```

- Interdisciplinary Connection Analysis: Analyze the response for mentions of concepts or ideas that could connect to other disciplines or courses.

```
prompt: |
  Identify any concepts or ideas in this student survey response that could
have
  interdisciplinary connections. For each connection, specify the related
discipline or course:
  '{{ input.response }}'
```

By breaking these tasks into separate operations, each LLM call can focus on a specific aspect of the analysis. This specialization might lead to more accurate results, depending on the LLM, data, and nature of the task!

How It Works

The DocETL optimizer operates using the following mechanism:

1. **Generation and Evaluation Agents**: These agents generate different plans for the pipeline according to predefined rewrite rules. Evaluation agents then compare plans and outputs to determine the best approach.
2. **Operator Rewriting**: The optimizer looks through operators in your pipeline where you've set optimize: true, and attempts to rewrite them using predefined rules.
3. **Output**: After optimization, DocETL outputs a new YAML file representing the optimized pipeline.

Using the Optimizer

You can invoke the optimizer using the following command:

```
docetl build your_pipeline.yaml
```

This command will save the optimized pipeline to `your_pipeline_opt.yaml`. Note that the optimizer will only rewrite operators where you've set `optimize: true`. Leaving this field unset will skip optimization for that operator.

Map Operation

The Map operation in DocETL applies a specified transformation to each item in your input data, allowing for complex processing and insight extraction from large, unstructured documents.

🚀 Example: Analyzing Long-Form News Articles

Let's see a practical example of using the Map operation to analyze long-form news articles, extracting key information and generating insights.

```yaml
- name: analyze_news_article
  type: map
  prompt: |
    Analyze the following news article:
    "{{ input.article }}"

    Provide the following information:
    1. Main topic (1-3 words)
    2. Summary (2-3 sentences)
    3. Key entities mentioned (list up to 5, with brief descriptions)
    4. Sentiment towards the main topic (positive, negative, or neutral)
    5. Potential biases or slants in reporting (if any)
    6. Relevant categories (e.g., politics, technology, environment; list up
to 3)
    7. Credibility score (1-10, where 10 is highly credible)

  output:
    schema:
      main_topic: string
      summary: string
      key_entities: list[object]
      sentiment: string
      biases: list[string]
      categories: list[string]
      credibility_score: integer

  model: gpt-4o-mini
  validate:
    - len(output["main_topic"].split()) <= 3
    - len(output["key_entities"]) <= 5
    - output["sentiment"] in ["positive", "negative", "neutral"]
    - len(output["categories"]) <= 3
    - 1 <= output["credibility_score"] <= 10
  num_retries_on_validate_failure: 2
```

This Map operation processes long-form news articles to extract valuable insights:

1. Identifies the main topic of the article.
2. Generates a concise summary.
3. Extracts key entities (people, organizations, locations) mentioned in the article.
4. Analyzes the overall sentiment towards the main topic.
5. Identifies potential biases or slants in the reporting.
6. Categorizes the article into relevant topics.
7. Assigns a credibility score based on the content and sources.

The operation includes validation to ensure the output meets our expectations and will retry up to 2 times if validation fails.

Sample Input and Output

Input:


```
[
  {
    "article": "In a groundbreaking move, the European Union announced
yesterday a comprehensive plan to transition all member states to 100%
renewable energy by 2050. The ambitious proposal, dubbed 'Green Europe 2050',
aims to completely phase out fossil fuels and nuclear power across the
continent.

    European Commission President Ursula von der Leyen stated, 'This is not
just about fighting climate change; it's about securing Europe's energy
independence and economic future.' The plan includes massive investments in
solar, wind, and hydroelectric power, as well as significant funding for
research into new energy storage technologies.

    However, the proposal has faced criticism from several quarters. Some
Eastern European countries, particularly Poland and Hungary, argue that the
timeline is too aggressive and could damage their economies, which are still
heavily reliant on coal. Industry groups have also expressed concern about the
potential for job losses in the fossil fuel sector.

    Environmental groups have largely praised the initiative, with Greenpeace
calling it 'a beacon of hope in the fight against climate change.' However,
some activists argue that the 2050 target is not soon enough, given the urgency
of the climate crisis.

    The plan also includes provisions for a 'just transition,' with billions of
euros allocated to retraining workers and supporting regions that will be most
affected by the shift away from fossil fuels. Additionally, it proposes
stricter energy efficiency standards for buildings and appliances, and
significant investments in public transportation and electric vehicle
infrastructure.

    Experts are divided on the feasibility of the plan. Dr. Maria Schmidt, an
energy policy researcher at the University of Berlin, says, 'While ambitious,
this plan is achievable with the right political will and technological
advancements.' However, Dr. John Smith from the London School of Economics
warns, 'The costs and logistical challenges of such a rapid transition should
not be underestimated.'

    As the proposal moves forward for debate in the European Parliament, it's
clear that 'Green Europe 2050' will be a defining issue for the continent in
the coming years, with far-reaching implications for Europe's economy,
environment, and global leadership in climate action."
  }
]
```


Output:


```
[
  {
    "main_topic": "EU Renewable Energy",
    "summary": "The European Union has announced a plan called 'Green Europe
2050' to transition all member states to 100% renewable energy by 2050. The
ambitious proposal aims to phase out fossil fuels and nuclear power, invest in
```

```
renewable energy sources, and includes provisions for a 'just transition' to
support affected workers and regions.",
    "key_entities": [
      {
        "name": "European Union",
        "description": "Political and economic union of 27 member states"
      },
      {
        "name": "Ursula von der Leyen",
        "description": "European Commission President"
      },
      {
        "name": "Poland",
        "description": "Eastern European country critical of the plan"
      },
      {
        "name": "Hungary",
        "description": "Eastern European country critical of the plan"
      },
      {
        "name": "Greenpeace",
        "description": "Environmental organization supporting the initiative"
      }
    ],
    "sentiment": "positive",
    "biases": [
      "Slight bias towards environmental concerns over economic impacts",
      "More emphasis on supportive voices than critical ones"
    ],
    "categories": [
      "Environment",
      "Politics",
      "Economy"
    ],
    "credibility_score": 8
  }
]
```


This example demonstrates how the Map operation can transform long, unstructured news articles into structured, actionable insights. These insights can be used for various purposes such as trend analysis, policy impact assessment, and public opinion monitoring.

Required Parameters

- `name`: A unique name for the operation.
- `type`: Must be set to "map".

Optional Parameters

Parameter	Description	Default
`prompt`	The prompt template to use for the transformation. Access input variables with `input.keyname`.	None
`batch_prompt`	Template for processing multiple documents in a single prompt. Access batch with `inputs` list.	None
`max_batch_size`	Maximum number of documents to process in a single batch	None
`output.schema`	Schema definition for the output from the LLM.	None
`output.n`	Number of outputs to generate for each input. (only available for OpenAI models; this is used to generate multiple outputs from a single input and automatically turn into a bigger list)	1
`model`	The language model to use	Falls back to `default_model`
`optimize`	Flag to enable operation optimization	`True`
`recursively_optimize`	Flag to enable recursive optimization of operators synthesized as part of rewrite rules	`false`
`sample`	Number of samples to use for the operation	Processes all data
`tools`	List of tool definitions for LLM use	None
`validate`	List of Python expressions to validate the output	None
`flush_partial_results`	Write results of individual batches of map operation to disk for faster inspection	False

Parameter	Description	Default
`num_retries_on_validate_failure`	Number of retry attempts on validation failure	0
`gleaning`	Configuration for advanced validation and LLM-based refinement	None
`drop_keys`	List of keys to drop from the input before processing	None
`timeout`	Timeout for each LLM call in seconds	120
`max_retries_per_timeout`	Maximum number of retries per timeout	2
`timeout`	Timeout for each LLM call in seconds	120
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls.	{}
`skip_on_error`	If true, skip the operation if the LLM returns an error.	False
`bypass_cache`	If true, bypass the cache for this operation.	False
`pdf_url_key`	If specified, the key in the input that contains the URL of the PDF to process.	None
`calibrate`	Improve consistency across documents by using sample data as reference anchors.	False
`num_calibration_docs`	Number of documents to use sample and generate outputs for, for calibration.	10

Note: If `drop_keys` is specified, `prompt` and `output` become optional parameters.

Validation and Gleaning

For more details on validation techniques and implementation, see operators.

Batch Processing

The Map operation supports processing multiple documents in a single prompt using the `batch_prompt` parameter. This can be more efficient than processing documents individually, especially for simpler tasks and shorter documents, especially when there are LLM call limits. However, larger batch sizes (even > 5) can lead to more incorrect results, so use this feature judiciously.

Batch Processing Example

```
- name: classify_documents
  type: map
  max_batch_size: 5  # Process up to 5 documents in a single LLM call
  batch_prompt: |
    Classify each of the following documents into categories (technology,
business, or science):

    {% for doc in inputs %}
    Document {{loop.index}}:
    {{doc.text}}
    {% endfor %}

    Provide a classification for each document.
  prompt: |
    Classify the following document:
    {{input.text}}
  output:
    schema:
      category: string
```

When using batch processing:

1. The `batch_prompt` template receives an `inputs` list containing the batch of documents
2. Use `max_batch_size` to control how many documents are processed in each batch
3. You must also provide a `prompt` parameter that will be used in case the batch prompt's response cannot be parsed into the output schema
4. Gleaning and validation are applied to each document in the batch individually, after the batch has been processed by the LLM

Batch Size Considerations

Choose your `max_batch_size` carefully:

- Larger batches may be more efficient but risk hitting token limits
- Start with smaller batches (3-5 documents) and adjust based on your needs
- Consider document length when setting batch size

Calibration for Consistency

The Map operation supports calibration to improve consistency across documents, especially for classification tasks or operations that require relative positioning (like rating scales). When enabled, calibration samples a subset of your documents, processes them with the original prompt, and then uses those results to generate reference anchors that help maintain consistency across all documents.

This is particularly useful for: - **Classification tasks** where documents need to be evaluated relative to each other - **Rating/scoring operations** where you want consistent scales - **Subjective judgments** that benefit from concrete examples

Document Priority Classification with Calibration

Imagine you're processing a large collection of customer support tickets and want to classify them by priority. Without calibration, the LLM might be inconsistent - a "medium" priority ticket early in processing might be classified as "high" later when the LLM sees more severe issues.

```yaml
- name: classify_ticket_priority
  type: map
  calibrate: true  # Enable calibration
  num_calibration_docs: 15  # Use 15 tickets for calibration
  prompt: |
    Classify the following customer support ticket by priority level:

    Subject: {{ input.subject }}
    Description: {{ input.description }}
    Customer Tier: {{ input.customer_tier }}

    Classify as: low, medium, high, or critical
  output:
    schema:
      priority: string
      reasoning: string
  model: gpt-4o-mini
```

How calibration works:

1. **Sample**: Randomly selects 15 tickets from your dataset (using seed=42 for reproducibility)
2. **Process**: Runs the original prompt on these 15 tickets
3. **Analyze**: An LLM analyzes the sample results and generates reference anchors
4. **Augment**: Appends these reference anchors to your original prompt
5. **Execute**: Processes all tickets with the augmented prompt

Example calibration output:

```
# Original prompt gets augmented with something like:
#
# For reference, consider 'Server completely down for 500+ users' → critical as
your baseline for critical issues.
# Documents similar to 'Login button not working for one user' → low priority.
# For reference, consider 'Payment processing delays affecting checkout' → high
as your standard for high priority issues.
```

When to Use Calibration

Calibration is most beneficial when:

- Your task requires relative judgments (rating scales, classifications)
- You're processing documents that vary widely in characteristics
- Consistency across the entire dataset is more important than individual accuracy
- You have enough data for meaningful sampling (at least 20+ documents)

Calibration Considerations

- Adds a small overhead cost (processes calibration samples + calibration analysis)
- Uses a fixed seed (42) for reproducible sampling
- The calibration LLM call uses temperature=0.0 for consistent results

Advanced Features

PDF Processing

The Map operation can directly process PDFs using Claude or Gemini models. To use this feature:

1. Your input dataset must contain a key representing the URL of the PDF to process
2. Specify this field name using the `pdf_url_key` parameter in your map operation
3. The URLs must be publicly accessible or accessible to your environment

PDF Processing Example

Here's an example of processing a dataset of papers, where each paper is represented by a URL.

```
datasets:
  papers:
    type: file
    path: "papers/urls.json"  # Contains documents with PDF URLs

default_model: gemini/gemini-2.0-flash  # or claude models
operations:
  - name: extract_paper_info
    type: map
    pdf_url_key: url  # Tells DocETL which field contains the PDF URL
    prompt: |
      Summarize the paper.
    output:
      schema:
        paper_info: string

pipeline:
  steps:
    - name: extract_paper_info
      input: papers
      operations:
        - extract_paper_info
```

Your input data (`papers/urls.json`) should contain documents with PDF URLs:

```
[
  {"url": "https://assets.anthropic.com/m/1cd9d098ac3e6467/original/Claude-3-Model-Card.pdf"},
  ...
]
```

DocETL will: 1. Download each PDF 2. Extract the text content 3. Pass the content to the LLM with your prompt 4. Return the processed results:

```
[
  {
    "url": "https://assets.anthropic.com/m/1cd9d098ac3e6467/original/Claude-3-Model-Card.pdf",
    "paper_info": "This paper introduces Claude 3.5 Haiku and the upgraded Claude 3.5 Sonnet..."
  },
  ...
]
```

Tool Use

Tools can extend the capabilities of the Map operation. Each tool is a Python function that can be called by the LLM during execution, and follows the OpenAI Function Calling API.

Tool Definition Example

```
tools:
- required: true
    code: |
    def count_words(text):
        return {"word_count": len(text.split())}
    function:
    name: count_words
    description: Count the number of words in a text string.
    parameters:
        type: object
        properties:
        text:
            type: string
        required:
        - text
```

Warning

Tool use and gleaning cannot be used simultaneously.

Input Truncation

If the input doesn't fit within the token limit, DocETL automatically truncates tokens from the middle of the input data, preserving the beginning and end which often contain more important context. A warning is displayed when truncation occurs.

Batching

If you have a really large collection of documents and you don't want to run them through the Map operation at the same time, you can use the `batch_size` parameter to process data in smaller chunks. This can significantly reduce memory usage and improve performance.

To enable batching in your map operations, you need to specify the `max_batch_size` parameter in your configuration.

```
- name: extract_summaries
  type: map
  max_batch_size: 5
  clustering_method: random
```

```
  prompt: |
    Summarize this text: "{{ input.text }}"
  output:
    schema:
      summary: string
```

In the above config, there will be no more than 5 API calls to the LLM at a time (i.e., 5 documents processed at a time, one per API call).

Dropping Keys

You can use a map operation to act as an LLM no-op, and just drop any key-value pairs you don't want to save to the output file. To do this, you can use the `drop_keys` parameter.

```
- name: drop_keys_example
  type: map
  drop_keys:
    - "keyname1"
    - "keyname2"
```

Best Practices

1. **Clear Prompts**: Write clear, specific prompts that guide the LLM to produce the desired output.
2. **Robust Validation**: Use validation to ensure output quality and consistency.
3. **Appropriate Model Selection**: Choose the right model for your task, balancing performance and cost.
4. **Optimize for Scale**: For large datasets, consider using `sample` to test your operation before running on the full dataset.
5. **Use Tools Wisely**: Leverage tools for complex calculations or operations that the LLM might struggle with. You can write any Python code in the tools, so you can even use tools to call other APIs or search the internet.

Synthetic Data Generation

The Map operation supports generating multiple outputs for each input using the `output.n` parameter. This is particularly useful for synthetic data generation, content variations, or when you need multiple alternatives for each input item.

When you set `output.n` to a value greater than 1, the operation will: 1. Process each input once 2. Generate multiple outputs based on the same input 3. Return all generated outputs as separate items in the result list

This multiplies your dataset size by the factor of `n`.

Synthetic Email Generation Example

Imagine you have a dataset of prospects and want to generate multiple email templates for each person. Here's how to generate 10 different email templates per prospect:

```yaml
datasets:
  prospects:
    type: file
    path: "prospects.json"  # Contains names, companies, roles, etc.

operations:
  - name: generate_email_templates
    type: map
    bypass_cache: true
    optimize: true
    output:
      n: 10  # Generate 10 unique emails per prospect
      schema:
        subject: "str"
        body: "str"
        call_to_action: "str"
    prompt: |
      Create a personalized cold outreach email for the following prospect:

      Name: {{ input.name }}
      Company: {{ input.company }}
      Role: {{ input.role }}
      Industry: {{ input.industry }}

      The email should:
      - Have a compelling subject line
      - Be brief (3-5 sentences)
      - Mention a specific pain point for their industry
      - Include a clear call to action
      - Sound natural and conversational, not sales-y

      Your response should be formatted as:
      Subject: [Your subject line]

      [Your email body]

      Call to action: [Your specific call to action]

pipeline:
  steps:
    - name: email_generation
      input: prospects
      operations:
        - generate_email_templates

  output:
    type: file
    path: "email_templates.json"
```

With this configuration, if your prospects.json file has 50 prospects, the output will contain 500 email templates (50 prospects × 10 emails each).

Important Considerations

- The `output.n` parameter is only available for OpenAI models
- Higher values of `n` will increase the cost of your operation proportionally
- For optimal results, keep your prompt focused and clear about generating diverse outputs
- When using pandas, the `n` parameter can be passed directly to the `map` method

Resolve Operation

The Resolve operation in DocETL identifies and canonicalizes duplicate entities in your data. It's particularly useful when dealing with inconsistencies that can arise from LLM-generated content, or data from multiple sources.

Motivation

Map operations executed by LLMs may sometimes yield inconsistent results, even when referring to the same entity. For example, when extracting patient names from medical transcripts, you might end up with variations like "Mrs. Smith" and "Jane Smith" for the same person. In such cases, a Resolve operation on the `patient_name` field can help standardize patient names before conducting further analysis.

🚀 Example: Standardizing Patient Names

Let's see a practical example of using the Resolve operation to standardize patient names extracted from medical transcripts.

```
- name: standardize_patient_names
  type: resolve
  optimize: true
  comparison_prompt: |
    Compare the following two patient name entries:

    Patient 1: {{ input1.patient_name }}
    Date of Birth 1: {{ input1.date_of_birth }}

    Patient 2: {{ input2.patient_name }}
    Date of Birth 2: {{ input2.date_of_birth }}

    Are these entries likely referring to the same patient? Consider name
similarity and date of birth. Respond with "True" if they are likely the same
patient, or "False" if they are likely different patients.
  resolution_prompt: |
    Standardize the following patient name entries into a single, consistent
format:

    {% for entry in inputs %}
    Patient Name {{ loop.index }}: {{ entry.patient_name }}
    {% endfor %}

    Provide a single, standardized patient name that represents all the
matched entries. Use the format "LastName, FirstName MiddleInitial" if
```

```
available.
  output:
    schema:
      patient_name: string
```

This Resolve operation processes patient names to identify and standardize duplicates:

1. Compares all pairs of patient names using the `comparison_prompt`. In the prompt, you can reference to the documents via `input1` and `input2`.
2. For identified duplicates, it applies the `resolution_prompt` to generate a standardized name. You can reference all matched entries via the `inputs` variable.

Note: The prompt templates use Jinja2 syntax, allowing you to reference input fields directly (e.g., `input1.patient_name`).

> ⚠ **Performance Consideration**
>
> You should not run this operation as-is unless your dataset is small! Running O(n^2) comparisons with an LLM can be extremely time-consuming for large datasets. Instead, optimize your pipeline first using `docetl build pipeline.yaml` and run the optimized version, which will generate efficient blocking rules for the operation. Make sure you've set `optimize: true` in your resolve operation config.

Blocking

To improve efficiency, the Resolve operation supports "blocking" - a technique to reduce the number of comparisons by only comparing entries that are likely to be matches. DocETL supports two types of blocking:

1. Embedding similarity: Compare embeddings of specified fields and only process pairs above a certain similarity threshold.
2. Python conditions: Apply custom Python expressions to determine if a pair should be compared.

Here's an example of a Resolve operation with blocking:

```
- name: standardize_patient_names
  type: resolve
  comparison_prompt: |
    # (Same as previous example)
  resolution_prompt: |
    # (Same as previous example)
  output:
    schema:
      patient_name: string
  blocking_keys:
```

```
    - last_name
    - date_of_birth
  blocking_threshold: 0.8
  blocking_conditions:
    - "left['last_name'][:2].lower() == right['last_name'][:2].lower()"
    - "left['first_name'][:2].lower() == right['first_name'][:2].lower()"
    - "left['date_of_birth'] == right['date_of_birth']"
    - "left['ssn'][-4:] == right['ssn'][-4:]"
```

In this example, pairs will be considered for comparison if:

- The embedding similarity of their `last_name` and `date_of_birth` fields is above 0.8, OR
- The `last_name` fields start with the same two characters, OR
- The `first_name` fields start with the same two characters, OR
- The `date_of_birth` fields match exactly, OR
- The last four digits of the `ssn` fields match.

How the Comparison Algorithm Works

After determining eligible pairs for comparison, the Resolve operation uses a Union-Find (Disjoint Set Union) algorithm to efficiently group similar items. Here's a breakdown of the process:

1. **Initialization**: Each item starts in its own cluster.
2. **Pair Generation**: All possible pairs of items are generated for comparison.
3. **Batch Processing**: Pairs are processed in batches (controlled by `compare_batch_size`).
4. **Comparison**: For each batch: a. An LLM performs pairwise comparisons to determine if items match. b. Matching pairs trigger a `merge_clusters` operation to combine their clusters.
5. **Iteration**: Steps 3-4 repeat until all pairs are compared.
6. **Result Collection**: All non-empty clusters are collected as the final result.

Efficiency

The batch processing of comparisons allows for efficient, incremental clustering as matches are found, without needing to rebuild the entire cluster structure after each match. This allows for parallelization of LLM calls, improving overall performance. However, this also limits parallelism to the batch size, so choose an appropriate value for `compare_batch_size` based on your dataset size and system capabilities.

Required Parameters

- `type`: Must be set to "resolve".
- `comparison_prompt`: The prompt template to use for comparing potential matches.
- `resolution_prompt`: The prompt template to use for reducing matched entries.
- `output`: Schema definition for the output from the LLM.

Optional Parameters

Parameter	Description	Default
`embedding_model`	The model to use for creating embeddings	Falls back to `default_model`
`resolution_model`	The language model to use for reducing matched entries	Falls back to `default_model`
`comparison_model`	The language model to use for comparing potential matches	Falls back to `default_model`
`blocking_keys`	List of keys to use for initial blocking	All keys in the input data
`blocking_threshold`	Embedding similarity threshold for considering entries as potential matches	None
`blocking_conditions`	List of conditions for initial blocking	[]
`input`	Specifies the schema or keys to subselect from each item to pass into the prompts	All keys from input items
`embedding_batch_size`	The number of entries to send to the embedding model at a time	1000
`compare_batch_size`	The number of entity pairs processed in each batch during the comparison phase	500

Parameter	Description	Default
`limit_comparisons`	Maximum number of comparisons to perform	None
`timeout`	Timeout for each LLM call in seconds	120
`max_retries_per_timeout`	Maximum number of retries per timeout	2
`sample`	Number of samples to use for the operation	None
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls.	{}
`bypass_cache`	If true, bypass the cache for this operation.	False

Best Practices

1. **Anticipate Resolve Needs**: If you anticipate needing a Resolve operation and want to control the prompts, create it in your pipeline and let the optimizer find the appropriate blocking rules and thresholds.
2. **Let the Optimizer Help**: The optimizer can detect if you need a Resolve operation (e.g., because there's a downstream reduce operation you're optimizing) and can create a Resolve operation with suitable prompts and blocking rules.
3. **Effective Comparison Prompts**: Design comparison prompts that consider all relevant factors for determining matches.
4. **Detailed Resolution Prompts**: Create resolution prompts that effectively standardize or combine information from matched records.
5. **Appropriate Model Selection**: Choose suitable models for embedding (if used) and language tasks.
6. **Optimize Batch Size**: If you expect to compare a large number of pairs, consider increasing the `compare_batch_size`. This parameter effectively limits parallelism, so a larger value can improve performance for large datasets.

Balancing Batch Size

While increasing `compare_batch_size` can improve parallelism, be cautious not to set it too high. Extremely large batch sizes might overwhelm system memory or exceed API rate limits. Consider your system's capabilities and the characteristics of your dataset when adjusting this parameter.

The Resolve operation is particularly useful for data cleaning, deduplication, and creating standardized records from multiple data sources. It can significantly improve data quality and consistency in your dataset.

Reduce Operation

The Reduce operation in DocETL aggregates data based on a key. It supports both batch reduction and incremental folding for large datasets, making it versatile for various data processing tasks.

Motivation

Reduce operations are essential when you need to summarize or aggregate data across multiple records. For example, you might want to:

- Analyze sentiment trends across social media posts
- Consolidate patient medical records from multiple visits
- Synthesize key findings from a set of research papers on a specific topic

🚀 Example: Summarizing Customer Feedback

Let's look at a practical example of using the Reduce operation to summarize customer feedback by department:

```
- name: summarize_feedback
  type: reduce
  reduce_key: department
  prompt: |
    Summarize the customer feedback for the {{ inputs[0].department }}
department:

    {% for item in inputs %}
    Feedback {{ loop.index }}: {{ item.feedback }}
    {% endfor %}

    Provide a concise summary of the main points and overall sentiment.
  output:
    schema:
      summary: string
      sentiment: string
```

This Reduce operation processes customer feedback grouped by department:

1. Groups all feedback entries by the 'department' key.
2. For each department, it applies the prompt to summarize the feedback and determine overall sentiment.

3. Outputs a summary and sentiment for each department.

Configuration

Required Parameters

- `type`: Must be set to "reduce".
- `reduce_key`: The key (or list of keys) to use for grouping data. Use `_all` to group all data into one group.
- `prompt`: The prompt template to use for the reduction operation.
- `output`: Schema definition for the output from the LLM.

Optional Parameters

Parameter	Description	Default
`sample`	Number of samples to use for the operation	None
`synthesize_resolve`	If false, won't synthesize a resolve operation between map and reduce	true
`model`	The language model to use	Falls back to default_model
`input`	Specifies the schema or keys to subselect from each item	All keys from input items
`pass_through`	If true, non-input keys from the first item in the group will be passed through	false
`associative`	If true, the reduce operation is associative (i.e., order doesn't matter)	true
`fold_prompt`	A prompt template for incremental folding	None
`fold_batch_size`	Number of items to process in each fold operation	None

Parameter	Description	Default
`value_sampling`	A dictionary specifying the sampling strategy for large groups	None
`verbose`	If true, enables detailed logging of the reduce operation	false
`persist_intermediates`	If true, persists the intermediate results for each group to the key `_{operation_name}_intermediates`	false
`timeout`	Timeout for each LLM call in seconds	120
`max_retries_per_timeout`	Maximum number of retries per timeout	2
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls.	{}
`bypass_cache`	If true, bypass the cache for this operation.	False

Advanced Features

Incremental Folding

For large datasets, the Reduce operation supports incremental folding. This allows processing of large groups in smaller batches, which can be more efficient and reduce memory usage.

To enable incremental folding, provide a `fold_prompt` and `fold_batch_size`:

```yaml
- name: large_data_reduce
  type: reduce
  reduce_key: category
  prompt: |
    Summarize the data for category {{ inputs[0].category }}:
    {% for item in inputs %}
    Item {{ loop.index }}: {{ item.data }}
    {% endfor %}
  fold_prompt: |
    Combine the following summaries for category {{ inputs[0].category }}:
    Current summary: {{ output.summary }}
```

```
    New data:
    {% for item in inputs %}
    Item {{ loop.index }}: {{ item.data }}
    {% endfor %}
  fold_batch_size: 100
  output:
    schema:
      summary: string
```

Example Rendered Prompt

Rendered Reduce Prompt

Let's consider an example of how a reduce operation prompt might look when rendered with actual data. Assume we have a reduce operation that summarizes product reviews, and we're processing reviews for a product with ID "PROD123". Here's what the rendered prompt might look like:

```
Summarize the reviews for product PROD123:

Review 1: This laptop is amazing! The battery life is incredible, lasting me a
full day of work without needing to charge. The display is crisp and vibrant,
perfect for both work and entertainment. The only minor drawback is that it can
get a bit warm during intensive tasks.

Review 2: I'm disappointed with this purchase. While the laptop looks sleek,
its performance is subpar. It lags when running multiple applications, and the
fan noise is quite noticeable. On the positive side, the keyboard is
comfortable to type on.

Review 3: Decent laptop for the price. It handles basic tasks well, but
struggles with more demanding software. The build quality is solid, and I
appreciate the variety of ports. Battery life is average, lasting about 6 hours
with regular use.

Review 4: Absolutely love this laptop! It's lightweight yet powerful, perfect
for my needs as a student. The touchpad is responsive, and the speakers produce
surprisingly good sound. My only wish is that it had a slightly larger screen.

Review 5: Mixed feelings about this product. The speed and performance are
great for everyday use and light gaming. However, the webcam quality is poor,
which is a letdown for video calls. The design is sleek, but the glossy finish
attracts fingerprints easily.
```

This example shows how the prompt template is filled with actual review data for a specific product. The language model would then process this prompt to generate a summary of the reviews for the product.

Scratchpad Technique

When doing an incremental reduce, the task may require intermediate state that is not represented in the output. For example, if the task is to determine all features more than

one person liked about the product, we need some intermediate state to keep track of the features that have been liked once, so if we see the same feature liked again, we can update the output. DocETL maintains an internal "scratchpad" to handle this.

How it works

1. The process starts with an empty accumulator and an internal scratchpad.
2. It sequentially folds in batches of more than one element at a time.
3. The internal scratchpad tracks additional state necessary for accurately solving tasks incrementally. The LLM decides what to write to the scratchpad.
4. During each internal LLM call, the current scratchpad state is used along with the accumulated output and new inputs.
5. The LLM updates both the accumulated output and the internal scratchpad, which are used in the next fold operation.

The scratchpad technique is handled internally by DocETL, allowing users to define complex reduce operations without worrying about the complexities of state management across batches. Users provide their reduce and fold prompts focusing on the desired output, while DocETL uses the scratchpad technique behind the scenes to ensure accurate tracking of trends and efficient processing of large datasets.

Value Sampling

For very large groups, you can use value sampling to process a representative subset of the data. This can significantly reduce processing time and costs.

The following table outlines the available value sampling methods:

Method	Description
random	Randomly select a subset of values
first_n	Select the first N values
cluster	Use K-means clustering to select representative samples
semantic_similarity	Select samples based on semantic similarity to a query

To enable value sampling, add a `value_sampling` configuration to your reduce operation. The configuration should specify the method, sample size, and any additional parameters required by the chosen method.

Value Sampling Configuration

```yaml
- name: sampled_reduce
  type: reduce
  reduce_key: product_id
  prompt: |
    Summarize the reviews for product {{ inputs[0].product_id }}:
    {% for item in inputs %}
    Review {{ loop.index }}: {{ item.review }}
    {% endfor %}
  value_sampling:
    enabled: true
    method: cluster
    sample_size: 50
  output:
    schema:
      summary: string
```

In this example, the Reduce operation will use K-means clustering to select a representative sample of 50 reviews for each product_id.

For semantic similarity sampling, you can use a query to select the most relevant samples. This is particularly useful when you want to focus on specific aspects of the data.

Semantic Similarity Sampling

```yaml
- name: sampled_reduce_sem_sim
  type: reduce
  reduce_key: product_id
  prompt: |
    Summarize the reviews for product {{ inputs[0].product_id }}, focusing on
comments about battery life and performance:
    {% for item in inputs %}
    Review {{ loop.index }}: {{ item.review }}
    {% endfor %}
  value_sampling:
    enabled: true
    method: sem_sim
    sample_size: 30
    embedding_model: text-embedding-3-small
    embedding_keys:
      - review
    query_text: "Battery life and performance"
  output:
    schema:
      summary: string
```

In this example, the Reduce operation will use semantic similarity to select the 30 reviews most relevant to battery life and performance for each product_id. This allows you to focus the summarization on specific aspects of the product reviews.

Lineage

The Reduce operation supports lineage, which allows you to track the original input data for each output. This can be useful for debugging and auditing. To enable lineage, add a `lineage` configuration to your reduce operation, specifying the keys to include in the lineage. For example:

```yaml
- name: summarize_reviews_by_category
  type: reduce
  reduce_key: category
  prompt: |
    Summarize the reviews for category {{ inputs[0].category }}:
    {% for item in inputs %}
    Review {{ loop.index }}: {{ item.review }}
    {% endfor %}
  output:
    schema:
      summary: string
    lineage:
      - product_id
```

This output will include a list of all product_ids for each category in the lineage, saved under the key `summarize_reviews_by_category_lineage`.

Best Practices

1. **Choose Appropriate Keys**: Select `reduce_key`(s) that logically group your data for the desired aggregation.
2. **Design Effective Prompts**: Create prompts that clearly instruct the model on how to aggregate or summarize the grouped data.
3. **Consider Data Size**: For large datasets, use incremental folding and value sampling to manage processing efficiently.
4. **Optimize Your Pipeline**: Use `docetl build pipeline.yaml` to optimize your pipeline, which can introduce efficient merge operations and resolve steps if needed.
5. **Balance Precision and Efficiency**: When dealing with very large groups, consider using value sampling to process a representative subset of the data.

Parallel Map Operation

The Parallel Map operation in DocETL applies multiple independent transformations to each item in the input data concurrently, maintaining a 1:1 input-to-output ratio while generating multiple fields simultaneously.

Similarity to Map Operation

The Parallel Map operation is very similar to the standard Map operation. The key difference is that Parallel Map allows you to define multiple prompts that run concurrently (without having to explicitly create a DAG), whereas a standard Map operation typically involves a single transformation.

Configuration

Each prompt in the Parallel Map operation is responsible for generating specific fields of the output. The prompts are executed concurrently, improving efficiency when working with multiple transformations.

The output schema should include all the fields generated by the individual prompts, ensuring that the results are combined into a single output item for each input.

Required Parameters

Parameter	Description
`name`	A unique name for the operation
`type`	Must be set to "parallel_map"
`prompts`	A list of prompt configurations (see below)
`output`	Schema definition for the combined output from all prompts

Each prompt configuration in the `prompts` list should contain:

- `prompt`: The prompt template to use for the transformation

- `output_keys` : List of keys that this prompt will generate
- `model` (optional): The language model to use for this specific prompt
- `gleaning` (optional): Advanced validation settings for this prompt (see Per-Prompt Gleaning section below)

Optional Parameters

Parameter	Description	Default
`model`	The default language model to use	Falls back to `default_model`
`optimize`	Flag to enable operation optimization	True
`recursively_optimize`	Flag to enable recursive optimization	false
`sample`	Number of samples to use for the operation	Processes all data
`timeout`	Timeout for each LLM call in seconds	120
`max_retries_per_timeout`	Maximum number of retries per timeout	2
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls.	{}

Why use Parallel Map instead of multiple Map operations?

While you could achieve similar results with multiple Map operations, Parallel Map offers several advantages:

1. **Concurrency**: Prompts run in parallel, potentially reducing overall processing time.
2. **Simplified Configuration**: You define multiple transformations in a single operation, reducing pipeline complexity.
3. **Unified Output**: Results from all prompts are combined into a single output item, simplifying downstream operations.

🚀 Example: Processing Job Applications

Here's an example of a parallel map operation that processes job applications by extracting key information and evaluating candidates:

```yaml
- name: process_job_application
  type: parallel_map
  prompts:
    - name: extract_skills
      prompt: "Given the following resume: '{{ input.resume }}', list the top
5 relevant skills for a software engineering position."
      output_keys:
        - skills
      gleaning:
        num_rounds: 1
        validation_prompt: |
          Confirm the skills list contains **exactly** 5 distinct skills and
each skill is one or two words long.
      model: gpt-4o-mini
    - name: calculate_experience
      prompt: "Based on the work history in this resume: '{{ input.resume }}',
calculate the total years of relevant experience for a software engineering
role."
      output_keys:
        - years_experience
      model: gpt-4o-mini
    - name: evaluate_cultural_fit
      prompt: "Analyze the following cover letter: '{{ input.cover_letter }}'.
Rate the candidate's potential cultural fit on a scale of 1-10, where 10 is
the highest."
      output_keys:
        - cultural_fit_score
      model: gpt-4o-mini
  output:
    schema:
      skills: list[string]
      years_experience: float
      cultural_fit_score: integer
```

This Parallel Map operation processes job applications by concurrently extracting skills, calculating experience, and evaluating cultural fit.

Advanced Validation: Per-Prompt Gleaning

Each prompt in a Parallel Map operation can include its own `gleaning` configuration. Gleaning works exactly as described in the operators overview but is **scoped to the individual LLM call** for that prompt. This allows you to tailor validation logic—and even the model used—to the specific transformation being performed.

The structure of the `gleaning` block is identical:

```yaml
gleaning:
  num_rounds: 1               # maximum refinement iterations
  validation_prompt: |        # judge prompt appended to the chat thread
    Ensure the extracted skills list contains at least 5 distinct items.
  model: gpt-4o-mini          # (optional) model for the validator LLM
```

Example with Per-Prompt Gleaning

```yaml
- name: process_job_application
  type: parallel_map
  prompts:
    - name: extract_skills
      prompt: "Given the following resume: '{{ input.resume }}', list the top
5 relevant skills for a software engineering position."
      output_keys:
        - skills
      gleaning:
        num_rounds: 1
        validation_prompt: |
          Confirm the skills list contains **exactly** 5 distinct skills and
each skill is one or two words long.
      model: gpt-4o-mini
    - name: calculate_experience
      prompt: "Based on the work history in this resume: '{{ input.resume }}',
calculate the total years of relevant experience for a software engineering
role."
      output_keys:
        - years_experience
      gleaning:
        num_rounds: 2
        validation_prompt: |
          Verify that the years of experience is a non-negative number and
round to one decimal place if necessary.
    - name: evaluate_cultural_fit
      prompt: "Analyze the following cover letter: '{{ input.cover_letter }}'.
Rate the candidate's potential cultural fit on a scale of 1-10, where 10 is
the highest."
      output_keys:
        - cultural_fit_score
      model: gpt-4o-mini
  output:
    schema:
      skills: list[string]
      years_experience: float
      cultural_fit_score: integer
```

In this configuration, only the `extract_skills` and `calculate_experience` prompts use gleaning. Each prompt's validator runs **immediately after** its own LLM call and before the overall outputs are merged.

Advantages

1. **Concurrency**: Multiple transformations are applied simultaneously, potentially reducing overall processing time.
2. **Simplicity**: Users can define multiple transformations without needing to create explicit DAGs in the configuration.
3. **Flexibility**: Different models can be used for different prompts within the same operation.
4. **Maintainability**: Each transformation can be defined and updated independently, making it easier to manage complex operations.

Best Practices

1. **Independent Transformations**: Ensure that the prompts in a Parallel Map operation are truly independent of each other to maximize the benefits of concurrent execution.
2. **Balanced Prompts**: Try to design prompts that have similar complexity and execution times to optimize overall performance.
3. **Output Schema Alignment**: Ensure that the output schema correctly captures all the fields generated by the individual prompts.
4. **Lightweight Validators**: When using per-prompt gleaning, keep validation prompts concise so that the cost and latency overhead stays manageable.

Filter Operation

The Filter operation in DocETL is used to selectively process data items based on specific conditions. It behaves similarly to the Map operation, but with a key difference: items that evaluate to false are filtered out of the dataset, allowing you to include or exclude data points from further processing in your pipeline.

Motivation

Filtering is crucial when you need to:

- Focus on the most relevant data points
- Remove noise or irrelevant information from your dataset
- Create subsets of data for specialized analysis
- Optimize downstream processing by reducing data volume

🚀 Example: Filtering High-Impact News Articles

Let's look at a practical example of using the Filter operation to identify high-impact news articles based on certain criteria.

```yaml
- name: filter_high_impact_articles
  type: filter
  prompt: |
    Analyze the following news article:
    Title: "{{ input.title }}"
    Content: "{{ input.content }}"

    Determine if this article is high-impact based on the following criteria:
    1. Covers a significant global or national event
    2. Has potential long-term consequences
    3. Affects a large number of people
    4. Is from a reputable source

    Respond with 'true' if the article meets at least 3 of these criteria,
otherwise respond with 'false'.

  output:
    schema:
      is_high_impact: boolean

  model: gpt-4-turbo
```

```
  validate:
    - isinstance(output["is_high_impact"], bool)
```

This Filter operation processes news articles and determines whether they are "high-impact" based on specific criteria. Unlike a Map operation, which would process all articles and add an "is_high_impact" field to each, this Filter operation will only pass through articles that meet the criteria, effectively removing low-impact articles from the dataset.

Sample Input and Output

Input:

```
[
  {
    "title": "Global Climate Summit Reaches Landmark Agreement",
    "content": "In a historic move, world leaders at the Global Climate Summit
have unanimously agreed to reduce carbon emissions by 50% by 2030. This
unprecedented agreement involves all major economies and sets binding targets
for renewable energy adoption, reforestation, and industrial emissions
reduction. Experts hail this as a turning point in the fight against climate
change, with potential far-reaching effects on global economies, energy
systems, and everyday life for billions of people."
  },
  {
    "title": "Local Bakery Wins Best Croissant Award",
    "content": "Downtown's favorite bakery, 'The Crusty Loaf', has been awarded
the title of 'Best Croissant' in the annual City Food Festival. Owner Maria
Garcia attributes the win to their use of imported French butter and a secret
family recipe. Local food critics praise the bakery's commitment to traditional
baking methods."
  }
]
```

Output:

```
[
  {
    "title": "Global Climate Summit Reaches Landmark Agreement",
    "content": "In a historic move, world leaders at the Global Climate Summit
have unanimously agreed to reduce carbon emissions by 50% by 2030. This
unprecedented agreement involves all major economies and sets binding targets
for renewable energy adoption, reforestation, and industrial emissions
reduction. Experts hail this as a turning point in the fight against climate
change, with potential far-reaching effects on global economies, energy
systems, and everyday life for billions of people."
  }
]
```

This example demonstrates how the Filter operation distinguishes between high-impact news articles and those of more local or limited significance. The climate summit article is retained in the dataset due to its global significance, long-term consequences, and

wide-ranging effects. The local bakery story, while interesting, doesn't meet the criteria for a high-impact article and is filtered out of the dataset.

Configuration

Required Parameters

- `name`: A unique name for the operation.
- `type`: Must be set to "filter".
- `prompt`: The prompt template to use for the filtering condition. Access input variables with `input.keyname`.
- `output`: Schema definition for the output from the LLM. It must include only one field, a boolean field.

Optional Parameters

See map optional parameters for additional configuration options, including `batch_prompt` and `max_batch_size`.

> **Validation**
>
> For more details on validation techniques and implementation, see operators.

Best Practices

1. **Clear Criteria**: Define clear and specific criteria for filtering in your prompt.
2. **Boolean Output**: Ensure your prompt guides the LLM to produce a clear boolean output.
3. **Data Flow Awareness**: Remember that unlike Map, Filter will reduce the size of your dataset. Ensure this aligns with your pipeline's objectives.

Equijoin Operation (Experimental)

The Equijoin operation in DocETL is an experimental feature designed for joining two datasets based on flexible, LLM-powered criteria. It leverages many of the same techniques as the Resolve operation, but applies them to the task of joining datasets rather than deduplicating within a single dataset.

Motivation

While traditional database joins rely on exact matches, real-world data often requires more nuanced joining criteria. Equijoin allows for joins based on semantic similarity or complex conditions, making it ideal for scenarios where exact matches are impossible or undesirable.

🚀 Example: Matching Job Candidates to Job Postings

Let's explore a practical example of using the Equijoin operation to match job candidates with suitable job postings based on skills and experience.

```
- name: match_candidates_to_jobs
  type: equijoin
  comparison_prompt: |
    Compare the following job candidate and job posting:

    Candidate Skills: {{ left.skills }}
    Candidate Experience: {{ left.years_experience }}

    Job Required Skills: {{ right.required_skills }}
    Job Desired Experience: {{ right.desired_experience }}

    Is this candidate a good match for the job? Consider both the overlap in
skills and the candidate's experience level. Respond with "True" if it's a
good match, or "False" if it's not a suitable match.
```

This Equijoin operation matches job candidates to job postings:

1. It uses the `comparison_prompt` to determine if a candidate is a good match for a job.
2. The operation can be optimized to use efficient blocking rules, reducing the number of comparisons.

Jinja2 Syntax with left and right

The prompt template uses Jinja2 syntax, allowing you to reference input fields directly (e.g., `left.skills`). You can reference the left and right documents using `left` and `right` respectively.

Performance Consideration

For large datasets, running comparisons with an LLM can be time-consuming. It's recommended to optimize your pipeline using `docetl build pipeline.yaml` to generate efficient blocking rules for the operation.

Blocking

Like the Resolve operation, Equijoin supports blocking techniques to improve efficiency. For details on how blocking works and how to implement it, please refer to the Blocking section in the Resolve operation documentation.

Adding Blocking Rules

Equijoin lets you specify **explicit blocking logic** to skip record pairs that are obviously unrelated *before* any LLM calls are made.

`blocking_keys`

Provide one or more **field names** for each side of the join. The selected values are concatenated and **embedded**; the cosine similarity of the left vs. right embeddings is then compared against `blocking_threshold` (defaults to `1.0`). If the similarity meets or exceeds that threshold, the pair moves on to the `comparison_prompt`; otherwise it is skipped.
If you omit `blocking_keys`, **all key–value pairs of each record are embedded by default**.

```yaml
blocking_keys:
  left:
    - medicine
  right:
    - extracted_medications
```

`blocking_threshold`

Optionally set a numeric `blocking_threshold` (0 – 1) representing the minimum cosine similarity (computed with the selected `embedding_model`) that the concatenated blocking

keys must achieve to be considered a candidate pair. Anything below the threshold is filtered out without invoking the LLM.

```
blocking_threshold: 0.35
embedding_model: text-embedding-3-small
```

A full Equijoin step combining both ideas might look like:

```
- name: join_meds_transcripts
  type: equijoin
  blocking_keys:
    left:
      - medicine
    right:
      - extracted_medications
  blocking_threshold: 0.3535
  embedding_model: text-embedding-3-small
  comparison_prompt: |
    Compare the following medication names:

    {{ left.medicine }}

    {{ right.extracted_medications }}

    Determine if these entries refer to the same medication.
```

Auto-generating Rules (Experimental)

`docetl build pipeline.yaml` can call the **Optimizer** to propose `blocking_keys` and an appropriate `blocking_threshold` based on a sample of your data. This feature is experimental; always review the suggested rules to ensure they do not exclude valid matches.

Parameters

Equijoin shares many parameters with the Resolve operation. For a detailed list of required and optional parameters, please see the Parameters section in the Resolve operation documentation.

Key differences for Equijoin include:

- `resolution_prompt` is not used in Equijoin.
- `limits` parameter is specific to Equijoin, allowing you to set maximum matches for each left and right item.

Incorporating Into a Pipeline

Here's an example of how to incorporate the Equijoin operation into a pipeline using the job candidate matching scenario:

```yaml
model: gpt-4o-mini

datasets:
  candidates:
    type: file
    path: /path/to/candidates.json
  job_postings:
    type: file
    path: /path/to/job_postings.json

operations:
  - name: match_candidates_to_jobs:
    type: equijoin
    comparison_prompt: |
      Compare the following job candidate and job posting:

      Candidate Skills: {{ left.skills }}
      Candidate Experience: {{ left.years_experience }}

      Job Required Skills: {{ right.required_skills }}
      Job Desired Experience: {{ right.desired_experience }}

      Is this candidate a good match for the job? Consider both the overlap in
skills and the candidate's experience level. Respond with "True" if it's a
good match, or "False" if it's not a suitable match.

pipeline:
  steps:
    - name: match_candidates_to_jobs
      operations:
        - match_candidates_to_jobs:
            left: candidates
            right: job_postings

  output:
    type: file
    path: "/path/to/matched_candidates_jobs.json"
```

This pipeline configuration demonstrates how to use the Equijoin operation to match job candidates with job postings. The pipeline reads candidate and job posting data from JSON files, performs the matching using the defined comparison prompt, and outputs the results to a new JSON file.

Best Practices

1. **Leverage the Optimizer**: Use `docetl build pipeline.yaml` to automatically generate efficient blocking rules for your Equijoin operation.

2. **Craft Thoughtful Comparison Prompts**: Design prompts that effectively determine whether two records should be joined based on your specific use case.
3. **Balance Precision and Recall**: When optimizing, consider the trade-off between catching all potential matches and reducing unnecessary comparisons.
4. **Mind Resource Constraints**: Use `limit_comparisons` if you need to cap the total number of comparisons for large datasets.
5. **Iterate and Refine**: Start with a small sample of your data to test and refine your join criteria before running on the full dataset.

For additional best practices that apply to both Resolve and Equijoin operations, see the Best Practices section in the Resolve operation documentation.

Rank Operation

The Rank operation in DocETL sorts documents based on specified criteria. Note that this operation is designed to sort documents along some (latent) attribute in the data. **It is not specifically meant for top-k or retrieval-like queries.**

We adapt algorithms from Human-Powered Sorts and Joins (VLDB 2012).

🚀 Example: Ranking Debates by Level of Controversy

Let's see a practical example of using the Rank operation to rank political debates based on how controversial they are:

```yaml
- name: rank_by_controversy
  type: rank
  prompt: |
    Order these debate transcripts based on how controversial the discussion
is.
    Consider factors like:
    - The level of disagreement between candidates
    - Discussion of divisive topics
    - Strong emotional language
    - Presence of conflicting viewpoints
    - Public reaction mentioned in the transcript

    Debates with the most controversial content should be ranked highest.
  input_keys: ["content", "title", "date"]
  direction: desc
  rerank_call_budget: 10 # max number of LLM calls to use; also optional
  initial_ordering_method: "likert"
```

This Rank operation ranks debate transcripts from most controversial to least controversial by:

1. First generating ordinal scores (on the Likert scale) for the ranking criteria and each document. This executes an LLM call **per document**.
2. Creating an initial ranking based on the scores from step 1.
3. Using an LLM to perform more precise re-rankings on a sliding window of documents. This executes `rerank_call_budget` calls.

Sample Input and Output

Input:


```
[
  {
    "title": "Presidential Debate: Economy and Trade",
    "date": "2020-09-29",
    "content": "Moderator: Let's discuss trade policies. Candidate A, your
response?\n\nCandidate A: My opponent's policies have shipped jobs overseas for
decades! Our workers are suffering while other countries laugh at
us.\n\nCandidate B: That's simply not true. The data shows our export growth
has been strong. My opponent doesn't understand basic economics.\n\nCandidate
A: [interrupting] You've been in government for 47 years and haven't fixed
anything!\n\nCandidate B: If you'd let me finish... The manufacturing sector
has actually added jobs under our policies.\n\nModerator: Please allow each
other to finish. Let's move to healthcare..."
  },
  {
    "title": "Vice Presidential Debate: Foreign Policy",
    "date": "2020-10-07",
    "content": "Moderator: What would your administration's approach be to
China?\n\nCandidate C: We need strategic engagement that protects American
interests while avoiding unnecessary conflict. My opponent has proposed
policies that would damage our diplomatic relationships.\n\nCandidate D: I
respectfully disagree with my colleague. Our current approach has been too
soft. We need to stand firm on human rights issues and trade
imbalances.\n\nCandidate C: I think we actually agree on the goals, if not the
methods. The question is how to achieve them without harmful
escalation.\n\nCandidate D: That's a fair point. Perhaps there's a middle
ground that maintains pressure while keeping dialogue open.\n\nModerator: Thank
you both for that thoughtful exchange. Moving to the Middle East..."
  }
]
```


Output:


```
[
  {
    "title": "Presidential Debate: Economy and Trade",
    "date": "2020-09-29",
    "content": "Moderator: Let's discuss trade policies. Candidate A, your
response?\n\nCandidate A: My opponent's policies have shipped jobs overseas for
decades! Our workers are suffering while other countries laugh at
us.\n\nCandidate B: That's simply not true. The data shows our export growth
has been strong. My opponent doesn't understand basic economics.\n\nCandidate
A: [interrupting] You've been in government for 47 years and haven't fixed
anything!\n\nCandidate B: If you'd let me finish... The manufacturing sector
has actually added jobs under our policies.\n\nModerator: Please allow each
other to finish. Let's move to healthcare...",
    "_rank": 1
  },
  {
    "title": "Vice Presidential Debate: Foreign Policy",
    "date": "2020-10-07",
    "content": "Moderator: What would your administration's approach be to
China?\n\nCandidate C: We need strategic engagement that protects American
interests while avoiding unnecessary conflict. My opponent has proposed
```

```
policies that would damage our diplomatic relationships.\n\nCandidate D: I
respectfully disagree with my colleague. Our current approach has been too
soft. We need to stand firm on human rights issues and trade
imbalances.\n\nCandidate C: I think we actually agree on the goals, if not the
methods. The question is how to achieve them without harmful
escalation.\n\nCandidate D: That's a fair point. Perhaps there's a middle
ground that maintains pressure while keeping dialogue open.\n\nModerator: Thank
you both for that thoughtful exchange. Moving to the Middle East...",
    "_rank": 2
  }
]
```

This example demonstrates how the Rank operation can semantically sort documents based on complex criteria, providing a ranking that would be difficult to achieve with keyword matching or rule-based approaches.

Algorithm and Implementation

The Rank operation works in these steps:

1. **Initial Ranking**:
    a. The algorithm begins with either an embedding-based or Likert-scale rating approach:
        i. **Embedding-based**: Creates embedding vectors for the ranking criteria and each document, then calculates cosine similarity
        ii. **Likert-based** (default): Uses the LLM to rate each document on a 7-point Likert scale based on the criteria. We do this in batches of `batch_size` documents (defaults to 10), and the prompt includes a random sample of `num_calibration_docs` (defaults to 10) documents to calibrate the LLM with.
    b. Documents are initially sorted by their similarity scores or ratings (high to low for desc, low to high for asc)
2. **"Picky Window" Refinement**:
    a. Rather than processing all documents with equal focus, the algorithm employs a "picky window" approach
    b. Starting from the bottom of the currently ranked documents and working upward:
        i. A large window of documents is presented to the LLM
        ii. The LLM is asked to identify only the top few documents (configured via `num_top_items_per_window`)
        iii. These chosen documents are then moved to the beginning of the window
    c. The window slides upward through the document set with overlapping segments

   d. This approach enables the algorithm to process many documents while focusing LLM effort on identifying the best matches

3. **Efficient Resource Utilization**:
   a. The window size and step size are calculated based on the call budget to ensure optimal use of LLM calls
   b. Overlap between windows ensures robust ranking with minimal redundancy
   c. The algorithm tracks document positions using unique identifiers to maintain consistency
4. **Output Preparation**:
   a. After all windows have been processed, the algorithm assigns a `_rank` field to each document (1-indexed)
   b. Returns the documents in their final sorted order

Required Parameters

- `name`: A unique name for the operation.
- `type`: Must be set to "rank".
- `prompt`: The prompt specifying the ranking criteria. This does **not** need to be a Jinja template.
- `input_keys`: List of document keys to consider for ranking.
- `direction`: Either "asc" (ascending) or "desc" (descending).

Optional Parameters

Parameter	Description	Default
`model`	The language model to use for LLM-based ranking	Falls back to `default_model`
`embedding_model`	The embedding model to use for similarity calculations	"text-embedding-3-small"
`batch_size`	Maximum number of documents to process in a single LLM batch rating (used for the first pass)	10

Parameter	Description	Default
`timeout`	Timeout for each LLM call in seconds	120
`verbose`	Whether to log detailed LLM call statistics	False
`num_calibration_docs`	Number of documents to use for calibration (used for the first pass)	10
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls	{}
`bypass_cache`	If true, bypass the cache for this operation	False
`initial_ordering_method`	Method to use for initial ranking: "likert" (default) or "embedding"	"likert"
`k`	Number of top items to focus on in the final ranking	None (ranks all items)
`call_budget`	Maximum number of LLM API calls to make during ranking	10
`num_top_items_per_window`	Number of top items the LLM should select from each window	3
`overlap_fraction`	Fraction of overlap between windows	0.5

Two-Step Ranking Approach

For more complex ranking tasks, a two-step approach can be more effective:

1. First use a `map` operation to extract and structure relevant information
2. Then use the `rank` operation to rank based on the extracted information

Two-Step Ranking Example

```
operations:
  - name: extract_hostile_exchanges
    type: map
    output:
      schema:
        meanness_summary: "str"
        hostility_level: "int"
        key_examples: "list[str]"
    prompt: |
      Analyze the following debate transcript for {{ input.title }} on {{
input.date }}:

      {{ input.content }}

      Extract and summarize exchanges where candidates are mean or hostile to
each other.
      [... prompt details ...]

  - name: rank_by_meanness
    type: rank
    prompt: |
      Order these debate transcripts based on how mean or hostile the
candidates are to each other.
      Focus on the meanness summaries and examples that have been extracted.

      Consider:
      - The overall hostility level rating
      - Severity of personal attacks in the key examples
      [... prompt details ...]
    input_keys: ["meanness_summary", "hostility_level", "key_examples",
"title", "date"]
    direction: desc
    rerank_call_budget: 10

pipeline:
  steps:
    - name: meanness_analysis
      input: debates
      operations:
        - extract_hostile_exchanges
        - rank_by_meanness
```

This approach: 1. First extracts structured data about hostility in each debate 2. Then ranks debates based on this pre-processed data

Best Practices

1. **Craft Clear Ranking Criteria**: Write clear, specific prompts that guide the LLM to understand the ranking priorities.

2. **Choose Appropriate Input Keys**: Only include document fields that are relevant to the ranking criteria to reduce noise.
3. **Consider Pre-Processing**: For complex criteria, use a map operation first to extract structured data that makes ranking more effective.
4. **Tune Window Parameters**:
5. Adjust `num_top_items_per_window` based on how selective you need the ranking to be
6. Modify `overlap_fraction` to balance redundancy and completeness
7. Start with defaults and adjust based on results
8. **Use Verbose Mode During Development**: Enable the `verbose` flag during development to understand how the ranking process works and verify the results.
9. **Direction Matters**: Choose "asc" or "desc" carefully based on your use case:
10. "desc" (descending) ranks the most matching items first
11. "asc" (ascending) ranks the least matching items first
12. **Mind Cost Considerations**: The ranking operation makes multiple LLM calls and embedding requests. For large datasets, consider sampling first to test your approach. Use the embedding based first pass to significantly reduce cost.

Performance Considerations

- The rank operation scales with O(n)
- The `verbose` flag adds detailed logging but doesn't affect performance or results

Extract Operation

Why use Extract instead of Map?

The Extract operation is specifically optimized for isolating portions of source text without synthesis or summarization. Unlike Map operations, which typically transform content, Extract pulls out specific sections verbatim. This provides three key advantages:

1. **Cost efficiency**: Lower output token costs when extracting large chunks of text
2. **Precision**: Extracts exact text without introducing LLM interpretation or potential hallucination
3. **Simplified workflow**: No need to define output schemas - extractions maintain the original text format

The Extract operation identifies and extracts specific sections of text from documents based on provided criteria. It's particularly useful for isolating relevant content from larger documents for further processing or analysis.

Example: Extracting Key Findings from Research Reports

Here's a practical example of using the Extract operation to pull out key findings from research reports:

```yaml
- name: findings
  type: extract
  prompt: |
    Extract all sections that discuss key findings, results, or conclusions
from this research report.
    Focus on paragraphs that:
    - Summarize experimental outcomes
    - Present statistical results
    - Describe discovered insights
    - State conclusions drawn from the research

    Only extract the most important and substantive findings.
  document_keys: ["report_text"]
  model: "gpt-4.1-mini"
```

This operation converts text into a line-numbered format, uses an LLM to identify relevant content, and extracts the specified text ranges. The extracted content is added to the document with the suffix "_extracted_findings".

Sample Input and Output

Input:


```
[
  {
    "report_id": "R-2023-001",
    "report_text": "EXPERIMENTAL METHODS\n\nThe study utilized a mixed-methods
approach combining quantitative surveys (n=230) and qualitative interviews
(n=42) with participants from diverse demographic backgrounds. Data collection
occurred between January and March 2023.\n\nRESULTS\n\nThe analysis revealed
three primary patterns of user engagement. First, 68% of participants reported
daily interaction with the platform, significantly higher than previous
industry benchmarks (p<0.01). Second, user retention showed strong correlation
with personalization features (r=0.72). Finally, demographic factors such as
age and technical proficiency were not significant predictors of engagement,
contradicting prior research in this domain.\n\nDISCUSSION\n\nThese findings
suggest that platform design priorities should emphasize personalization
capabilities over demographic targeting. The high daily engagement rates
indicate market readiness for similar applications, while the lack of
demographic effects points to broad accessibility across user
segments.\n\nLIMITATIONS\n\nThe study was limited by its focus on early
adopters, which may not represent the broader potential user base.
Additionally, the three-month timeframe may not capture seasonal variations in
user behavior."
  }
]
```


Output:


```
[
  {
    "report_id": "R-2023-001",
    "report_text": "EXPERIMENTAL METHODS\n\nThe study utilized a mixed-methods
approach combining quantitative surveys (n=230) and qualitative interviews
(n=42) with participants from diverse demographic backgrounds. Data collection
occurred between January and March 2023.\n\nRESULTS\n\nThe analysis revealed
three primary patterns of user engagement. First, 68% of participants reported
daily interaction with the platform, significantly higher than previous
industry benchmarks (p<0.01). Second, user retention showed strong correlation
with personalization features (r=0.72). Finally, demographic factors such as
age and technical proficiency were not significant predictors of engagement,
contradicting prior research in this domain.\n\nDISCUSSION\n\nThese findings
suggest that platform design priorities should emphasize personalization
capabilities over demographic targeting. The high daily engagement rates
indicate market readiness for similar applications, while the lack of
demographic effects points to broad accessibility across user
segments.\n\nLIMITATIONS\n\nThe study was limited by its focus on early
adopters, which may not represent the broader potential user base.
Additionally, the three-month timeframe may not capture seasonal variations in
user behavior.",
    "report_text_extracted_findings": "The analysis revealed three primary
patterns of user engagement. First, 68% of participants reported daily
interaction with the platform, significantly higher than previous industry
benchmarks (p<0.01). Second, user retention showed strong correlation with
personalization features (r=0.72). Finally, demographic factors such as age and
technical proficiency were not significant predictors of engagement,
contradicting prior research in this domain.\n\nThese findings suggest that
```

```
  platform design priorities should emphasize personalization capabilities over
  demographic targeting. The high daily engagement rates indicate market
  readiness for similar applications, while the lack of demographic effects
  points to broad accessibility across user segments."
    }
  ]
```

Output Formats

The Extract operation offers two output formats controlled by the `format_extraction` parameter:

String Format (Default)

With `format_extraction: true`, extracted text segments are joined with newlines into a single string:

```yaml
- name: findings
  type: extract
  prompt: "Extract the key findings from this research report."
  document_keys: ["report_text"]
  format_extraction: true  # Default setting
```

The resulting output combines all extractions:

```
{
  "report_id": "R-2023-001",
  "report_text": "... original text ...",
  "report_text_extracted_findings": "Finding 1 about daily engagement
rates...\n\nFinding 2 about personalization features..."
}
```

This format works well for human readability, further LLM processing, and when extractions should be treated as a coherent unit.

List Format

With `format_extraction: false`, each extracted text segment remains separate in a list:

```yaml
- name: findings
  type: extract
  prompt: "Extract the key findings from this research report."
  document_keys: ["report_text"]
  format_extraction: false
```

The resulting output preserves each extraction as a distinct item:

```
{
  "report_id": "R-2023-001",
  "report_text": "... original text ...",
  "report_text_extracted_findings": [
    "Finding 1 about daily engagement rates...",
    "Finding 2 about personalization features..."
  ]
}
```

This format is better for individual processing of extractions, counting distinct items, or creating structured data from separate extractions.

Extraction Strategies

The Extract operation offers two main strategies:

Line Number Strategy

This strategy reformats the input text with line numbers, then asks the LLM to identify relevant line ranges. The system extracts those specific ranges, removes line number prefixes, and eliminates duplicates. This works well for extracting multi-line passages or entire sections.

Regex Strategy

This strategy asks the LLM to generate regex patterns matching the desired content. The system applies these patterns to find matches in the original text. This works well for extracting structured data like dates, codes, or specific formatted information.

Required Parameters

- `name`: Unique name for the operation
- `type`: Must be "extract"
- `prompt`: Instructions specifying what content to extract. This does **not** need to be a Jinja template.
- `document_keys`: List of document fields containing text to process

Optional Parameters

Parameter	Description	Default
`model`	Language model to use	Falls back to `default_model`
`extraction_method`	"line_number" or "regex"	"line_number"
`format_extraction`	Join with newlines (`true`) or keep as list (`false`)	`true`
`extraction_key_suffix`	Suffix for output field names	"*extracted*"
`timeout`	Timeout for LLM calls in seconds	120
`skip_on_error`	Continue processing if errors occur	false
`litellm_completion_kwargs`	Additional parameters for LiteLLM calls	{}

Best Practices

Create specific extraction prompts with clear criteria about what content to extract or exclude. Choose the appropriate extraction method based on your needs:

- Use `line_number` for extracting paragraphs, sections, or content spanning multiple lines
- Use `regex` for extracting specific patterns like dates, codes, or formatted data

Process only document fields containing relevant text, and consider enabling `skip_on_error` for batch processing where individual failures shouldn't halt the entire operation.

Format your output based on downstream requirements: - Use the default string format when the extractions form a coherent whole - Use the list format when each extraction needs individual processing or analysis

Use Cases

The Extract operation is valuable for:

- Document summarization - extracting executive summaries or key findings
- Data extraction - isolating dates, measurements, or specific values from text
- Content filtering - pulling relevant sections from lengthy documents
- Evidence gathering - collecting specific statements on given topics
- Preprocessing - creating focused inputs for downstream analysis

Cluster operation

The Cluster operation in DocETL groups all items into a binary tree using agglomerative clustering of the embedding of some keys, and annotates each item with the path through this tree down to the item (Note that the path is reversed, starting with the most specific grouping, and ending in the root of the tree, the cluster that encompasses all your input).

Each cluster is summarized using an llm prompt, taking the summaries of its children as inputs (or for the leaf nodes, the actual items).

🚀 Example: Grouping concepts from a knowledge-graph

```yaml
- name: cluster_concepts
  type: cluster
  max_batch_size: 5
  embedding_keys:
    - concept
    - description
  output_key: categories # This is optional, and defaults to "clusters"
  summary_schema:
    concept: str
    description: str
  summary_prompt: |
    The following describes two related concepts. What concept
    encompasses both? Try not to be too broad; it might be that one of
    these two concepts already encompasses the other; in that case,
    you should just use that concept.

    {% for input in inputs %}
    {{input.concept}}:
    {{input.description}}
    {% endfor %}

    Provide the title of the super-concept, and a description.
```

This cluster operation processes a set of concepts, each with a title and a description, and groups them into a tree of categories.

Sample Input and Output

Input:


```
[
  {
    "concept": "Shed",
    "description": "A shed is typically a simple, single-story roofed
structure, often used for storage, for hobbies, or as a workshop, and typically
serving as outbuilding, such as in a back garden or on an allotment. Sheds vary
considerably in their size and complexity of construction, from simple open-
sided ones designed to cover bicycles or garden items to large wood-framed
structures with shingled roofs, windows, and electrical outlets. Sheds used on
farms or in the industry can be large structures. The main types of shed
construction are metal sheathing over a metal frame, plastic sheathing and
frame, all-wood construction (the roof may be asphalt shingled or sheathed in
tin), and vinyl-sided sheds built over a wooden frame. Small sheds may include
a wooden or plastic floor, while more permanent ones may be built on a concrete
pad or foundation. Sheds may be lockable to deter theft or entry by children,
domestic animals, wildlife, etc."
  },
  {
    "concept": "Barn",
    "description": "A barn is an agricultural building usually on farms and
used for various purposes. In North America, a barn refers to structures that
house livestock, including cattle and horses, as well as equipment and fodder,
and often grain.[2] As a result, the term barn is often qualified e.g. tobacco
barn, dairy barn, cow house, sheep barn, potato barn. In the British Isles, the
term barn is restricted mainly to storage structures for unthreshed cereals and
fodder, the terms byre or shippon being applied to cow shelters, whereas horses
are kept in buildings known as stables.[2][3] In mainland Europe, however,
barns were often part of integrated structures known as byre-dwellings (or
housebarns in US literature). In addition, barns may be used for equipment
storage, as a covered workplace, and for activities such as threshing."
  },
  {
    "concept": "Tree house",
    "description": "A tree house, tree fort or treeshed, is a platform or
building constructed around, next to or among the trunk or branches of one or
more mature trees while above ground level. Tree houses can be used for
recreation, work space, habitation, a hangout space and observation. People
occasionally connect ladders or staircases to get up to the platforms."
  },
  {
    "concept": "Castle",
    "description": "A castle is a type of fortified structure built during the
Middle Ages predominantly by the nobility or royalty and by military orders.
Scholars usually consider a castle to be the private fortified residence of a
lord or noble. This is distinct from a mansion, palace, and villa, whose main
purpose was exclusively for pleasance and are not primarily fortresses but may
be fortified.[a] Use of the term has varied over time and, sometimes, has also
been applied to structures such as hill forts and 19th- and 20th-century homes
built to resemble castles. Over the Middle Ages, when genuine castles were
built, they took on a great many forms with many different features, although
some, such as curtain walls, arrowslits, and portcullises, were commonplace."
  },
  {
    "concept": "Fortress",
    "description": "A fortification (also called a fort, fortress, fastness, or
```

```
stronghold) is a military construction designed for the defense of territories
in warfare, and is used to establish rule in a region during peacetime. The
term is derived from Latin fortis ('strong') and facere ('to make'). From very
early history to modern times, defensive walls have often been necessary for
cities to survive in an ever-changing world of invasion and conquest. Some
settlements in the Indus Valley Civilization were the first small cities to be
fortified. In ancient Greece, large stone walls had been built in Mycenaean
Greece, such as the ancient site of Mycenae (known for the huge stone blocks of
its 'cyclopean' walls). A Greek phrourion was a fortified collection of
buildings used as a military garrison, and is the equivalent of the Roman
castellum or fortress. These constructions mainly served the purpose of a watch
tower, to guard certain roads, passes, and borders. Though smaller than a real
fortress, they acted as a border guard rather than a real strongpoint to watch
and maintain the border."
  }
]
```

Output:


```
[
  {
    "concept": "Shed",
    "description": "A shed is typically a simple, single-story roofed
structure, often used for storage, for hobbies, or as a workshop, and typically
serving as outbuilding, such as in a back garden or on an allotment. Sheds vary
considerably in their size and complexity of construction, from simple open-
sided ones designed to cover bicycles or garden items to large wood-framed
structures with shingled roofs, windows, and electrical outlets. Sheds used on
farms or in the industry can be large structures. The main types of shed
construction are metal sheathing over a metal frame, plastic sheathing and
frame, all-wood construction (the roof may be asphalt shingled or sheathed in
tin), and vinyl-sided sheds built over a wooden frame. Small sheds may include
a wooden or plastic floor, while more permanent ones may be built on a concrete
pad or foundation. Sheds may be lockable to deter theft or entry by children,
domestic animals, wildlife, etc.",
    "categories": [
      {
        "distance": 0.9907871670904073,
        "concept": "Outbuildings",
        "description": "Outbuildings are structures that are separate from a
main building, typically located on a property for purposes such as storage,
workshops, or housing animals and equipment. This category includes structures
like sheds and barns, which serve specific functions like storing tools,
equipment, or livestock."
      },
      {
        "distance": 1.148880974178631,
        "concept": "Auxiliary Structures",
        "description": "Auxiliary structures are secondary or additional
buildings that serve various practical purposes related to a main dwelling or
property. This category encompasses structures like tree houses and
outbuildings, which provide functional, recreational, or storage spaces, often
designed to enhance the usability of the property."
      },
      {
        "distance": 1.292957924480073,
        "concept": "Military and Support Structures",
        "description": "Military and support structures refer to various types
of constructions designed for specific functions related to defense and
utility. This concept encompasses fortified structures, such as castles and
```

```
fortresses, built for protection and military purposes, as well as auxiliary
structures that serve practical roles for main buildings, including storage,
recreation, and additional facilities. Together, these structures enhance the
safety, functionality, and usability of a property or territory."
      }
    ]
  },
  {
    "concept": "Barn",
    "description": "A barn is an agricultural building usually on farms and
used for various purposes. In North America, a barn refers to structures that
house livestock, including cattle and horses, as well as equipment and fodder,
and often grain.[2] As a result, the term barn is often qualified e.g. tobacco
barn, dairy barn, cow house, sheep barn, potato barn. In the British Isles, the
term barn is restricted mainly to storage structures for unthreshed cereals and
fodder, the terms byre or shippon being applied to cow shelters, whereas horses
are kept in buildings known as stables.[2][3] In mainland Europe, however,
barns were often part of integrated structures known as byre-dwellings (or
housebarns in US literature). In addition, barns may be used for equipment
storage, as a covered workplace, and for activities such as threshing.",
    "categories": [
      {
        "distance": 0.9907871670904073,
        "concept": "Outbuildings",
        "description": "Outbuildings are structures that are separate from a
main building, typically located on a property for purposes such as storage,
workshops, or housing animals and equipment. This category includes structures
like sheds and barns, which serve specific functions like storing tools,
equipment, or livestock."
      },
      {
        "distance": 1.148880974178631,
        "concept": "Auxiliary Structures",
        "description": "Auxiliary structures are secondary or additional
buildings that serve various practical purposes related to a main dwelling or
property. This category encompasses structures like tree houses and
outbuildings, which provide functional, recreational, or storage spaces, often
designed to enhance the usability of the property."
      },
      {
        "distance": 1.292957924480073,
        "concept": "Military and Support Structures",
        "description": "Military and support structures refer to various types
of constructions designed for specific functions related to defense and
utility. This concept encompasses fortified structures, such as castles and
fortresses, built for protection and military purposes, as well as auxiliary
structures that serve practical roles for main buildings, including storage,
recreation, and additional facilities. Together, these structures enhance the
safety, functionality, and usability of a property or territory."
      }
    ]
  },
  {
    "concept": "Tree house",
    "description": "A tree house, tree fort or treeshed, is a platform or
building constructed around, next to or among the trunk or branches of one or
more mature trees while above ground level. Tree houses can be used for
recreation, work space, habitation, a hangout space and observation. People
occasionally connect ladders or staircases to get up to the platforms.",
    "categories": [
      {
```

```
        "distance": 1.148880974178631,
        "concept": "Auxiliary Structures",
        "description": "Auxiliary structures are secondary or additional
buildings that serve various practical purposes related to a main dwelling or
property. This category encompasses structures like tree houses and
outbuildings, which provide functional, recreational, or storage spaces, often
designed to enhance the usability of the property."
      },
      {
        "distance": 1.292957924480073,
        "concept": "Military and Support Structures",
        "description": "Military and support structures refer to various types
of constructions designed for specific functions related to defense and
utility. This concept encompasses fortified structures, such as castles and
fortresses, built for protection and military purposes, as well as auxiliary
structures that serve practical roles for main buildings, including storage,
recreation, and additional facilities. Together, these structures enhance the
safety, functionality, and usability of a property or territory."
      }
    ]
  },
  {
    "concept": "Castle",
    "description": "A castle is a type of fortified structure built during the
Middle Ages predominantly by the nobility or royalty and by military orders.
Scholars usually consider a castle to be the private fortified residence of a
lord or noble. This is distinct from a mansion, palace, and villa, whose main
purpose was exclusively for pleasance and are not primarily fortresses but may
be fortified.[a] Use of the term has varied over time and, sometimes, has also
been applied to structures such as hill forts and 19th- and 20th-century homes
built to resemble castles. Over the Middle Ages, when genuine castles were
built, they took on a great many forms with many different features, although
some, such as curtain walls, arrowslits, and portcullises, were commonplace.",
    "categories": [
      {
        "distance": 0.9152435235428339,
        "concept": "Fortified structures",
        "description": "Fortified structures refer to buildings designed to
protect from attacks and enhance defense. This category encompasses various
forms of military architecture, including castles and fortresses. Castles serve
as private residences for nobility or military orders with substantial
fortification features, while fortresses are broader military constructions
aimed at defending territories and establishing control. Both types share the
common purpose of defense against invasion, though they serve different social
and functional roles."
      },
      {
        "distance": 1.292957924480073,
        "concept": "Military and Support Structures",
        "description": "Military and support structures refer to various types
of constructions designed for specific functions related to defense and
utility. This concept encompasses fortified structures, such as castles and
fortresses, built for protection and military purposes, as well as auxiliary
structures that serve practical roles for main buildings, including storage,
recreation, and additional facilities. Together, these structures enhance the
safety, functionality, and usability of a property or territory."
      }
    ]
  },
  {
    "concept": "Fortress",
```

```
    "description": "A fortification (also called a fort, fortress, fastness, or
stronghold) is a military construction designed for the defense of territories
in warfare, and is used to establish rule in a region during peacetime. The
term is derived from Latin fortis ('strong') and facere ('to make'). From very
early history to modern times, defensive walls have often been necessary for
cities to survive in an ever-changing world of invasion and conquest. Some
settlements in the Indus Valley Civilization were the first small cities to be
fortified. In ancient Greece, large stone walls had been built in Mycenaean
Greece, such as the ancient site of Mycenae (known for the huge stone blocks of
its 'cyclopean' walls). A Greek phrourion was a fortified collection of
buildings used as a military garrison, and is the equivalent of the Roman
castellum or fortress. These constructions mainly served the purpose of a watch
tower, to guard certain roads, passes, and borders. Though smaller than a real
fortress, they acted as a border guard rather than a real strongpoint to watch
and maintain the border.",
    "categories": [
      {
        "distance": 0.9152435235428339,
        "concept": "Fortified structures",
        "description": "Fortified structures refer to buildings designed to
protect from attacks and enhance defense. This category encompasses various
forms of military architecture, including castles and fortresses. Castles serve
as private residences for nobility or military orders with substantial
fortification features, while fortresses are broader military constructions
aimed at defending territories and establishing control. Both types share the
common purpose of defense against invasion, though they serve different social
and functional roles."
      },
      {
        "distance": 1.292957924480073,
        "concept": "Military and Support Structures",
        "description": "Military and support structures refer to various types
of constructions designed for specific functions related to defense and
utility. This concept encompasses fortified structures, such as castles and
fortresses, built for protection and military purposes, as well as auxiliary
structures that serve practical roles for main buildings, including storage,
recreation, and additional facilities. Together, these structures enhance the
safety, functionality, and usability of a property or territory."
      }
    ]
  }
]
```


Required Parameters

- `name`: A unique name for the operation.
- `type`: Must be set to "cluster".
- `embedding_keys`: A list of keys to use for the embedding that is clustered on
- `summary_prompt`: The prompt used to summarize a cluster based on its children. Access input variables by iterating over `inputs` with `{% for input in inputs %}` and accessing properties with `{{input.keyname}}`.
- `summary_schema`: The schema for the summary of each cluster. This is the output schema for the `summary_prompt` based llm call.

Optional Parameters

Parameter	Description	Default
`output_key`	The name of the output key where the cluster path will be inserted in the items.	"clusters"
`model`	The language model to use	Falls back to `default_model`
`embedding_model`	The embedding model to use	"text-embedding-3-small"
`timeout`	Timeout for each LLM call in seconds	120
`max_retries_per_timeout`	Maximum number of retries per timeout	2
`sample`	Number of items to sample for this operation	None
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls.	{}

Split Operation

The Split operation in DocETL is designed to divide long text content into smaller, manageable chunks. This is particularly useful when dealing with large documents that exceed the token limit of language models or when the LLM's performance degrades with increasing input size for complex tasks.

Motivation

Some common scenarios where the Split operation is valuable include:

- Processing long customer support transcripts to analyze specific sections
- Dividing extensive research papers or reports for detailed analysis
- Breaking down large legal documents to extract relevant clauses or sections
- Preparing long-form content for summarization or topic extraction

🚀 Operation Example: Splitting Customer Support Transcripts

Here's an example of using the Split operation to divide customer support transcripts into manageable chunks:

```yaml
- name: split_transcript
  type: split
  split_key: transcript
  method: token_count
  method_kwargs:
    num_tokens: 500
    model: gpt-4o-mini
```

This Split operation processes long customer support transcripts:

1. Splits the 'transcript' field into chunks of approximately 500 tokens each.
2. Uses the gpt-4o-mini model's tokenizer for accurate token counting.
3. Generates multiple output items for each input item, one for each chunk.

Note that chunks will not overlap in content.

Configuration

Required Parameters

- `type`: Must be set to "split".
- `split_key`: The key of the field containing the text to split.
- `method`: The method to use for splitting. Options are "delimiter" and "token_count".
- `method_kwargs`: A dictionary of keyword arguments for the splitting method.
- For "delimiter" method: `delimiter` (string) to use for splitting.
- For "token_count" method: `num_tokens` (integer) specifying the maximum number of tokens per chunk.

Optional Parameters in `method_kwargs

Parameter	Description	Default
`model`	The language model's tokenizer to use	Falls back to `default_model`
`num_splits_to_group`	Number of splits to group together into one chunk (only for "delimiter" method)	1
`sample`	Number of samples to use for the operation	None

Splitting Methods

Token Count Method

The token count method splits the text into chunks based on a specified number of tokens. This is useful when you need to ensure that each chunk fits within the token limit of your language model, or you know that smaller chunks lead to higher performance.

Delimiter Method

The delimiter method splits the text based on a specified delimiter string. This is particularly useful when you want to split your text at logical boundaries, such as paragraphs or sections.

Delimiter Method Example

If you set the `delimiter` to `"\n\n"` (double newline) and `num_splits_to_group` to 3, each chunk will contain 3 paragraphs.

```
- name: split_by_paragraphs
  type: split
  split_key: document
  method: delimiter
  method_kwargs:
    delimiter: "\n\n"
  num_splits_to_group: 3
```

Output

The Split operation generates multiple output items for each input item:

- All original key-value pairs from the input item.
- `{split_key}_chunk`: The content of the split chunk.
- `{op_name}_id`: A unique identifier for each original document.
- `{op_name}_chunk_num`: The sequential number of the chunk within its original document.

Use Cases

1. **Analyzing Customer Frustration**: Split long support transcripts, then use a map operation to identify frustration indicators in each chunk, followed by a reduce operation to summarize frustration points across the chunks (per transcript).
2. **Document Summarization**: Split large documents, apply a map operation for section-wise summarization, then use a reduce operation to compile an overall summary.
3. **Topic Extraction from Research Papers**: Divide research papers into sections, use a map operation to extract key topics from each section, then apply a reduce operation to synthesize main themes across the entire paper.

🚀 End-to-End Pipeline Example: Analyzing Customer Frustration

Let's walk through a complete example of using Split, Map, and Reduce operations to analyze customer frustration in support transcripts.

Step 1: Split Operation

```yaml
- name: split_transcript
  type: split
  split_key: transcript
  method: token_count
  method_kwargs:
    num_tokens: 500
    model: gpt-4o-mini
```

Step 2: Map Operation (Identify Frustration Indicators)

```yaml
- name: identify_frustration
  type: map
  input:
    - transcript_chunk
  prompt: |
    Analyze the following customer support transcript chunk for signs of
customer frustration:

    {{ input.transcript_chunk }}

    Identify any indicators of frustration, such as:
    1. Use of negative language
    2. Repetition of issues
    3. Expressions of dissatisfaction
    4. Requests for escalation

    Provide a list of frustration indicators found, if any.
  output:
    schema:
      frustration_indicators: list[string]
```

Step 3: Reduce Operation (Summarize Frustration Points)

```yaml
- name: summarize_frustration
  type: reduce
  reduce_key: split_transcript_id
  associative: false
  prompt: |
    Summarize the customer frustration points for this support transcript:

    {% for item in inputs %}
    Chunk {{ item.split_transcript_chunk_num }}:
    {% for indicator in item.frustration_indicators %}
    - {{ indicator }}
    {% endfor %}
    {% endfor %}

    Provide a concise summary of the main frustration points and their
frequency or intensity across the entire transcript.
  output:
```

```
    schema:
      frustration_summary: string
      primary_issues: list[string]
      frustration_level: string # e.g., "low", "medium", "high"
```

Non-Associative Reduce Operation

Note the `associative: false` parameter in the reduce operation. This is crucial when the order of the chunks matters for your analysis. It ensures that the reduce operation processes the chunks in the order they appear in the original transcript, which is often important for understanding the context and progression of customer frustration.

Explanation

1. The **Split** operation divides long transcripts into 500-token chunks.
2. The **Map** operation analyzes each chunk for frustration indicators.
3. The **Reduce** operation combines the frustration indicators from all chunks of a transcript, summarizing the overall frustration points, primary issues, and assessing the overall frustration level. The `associative: false` setting ensures that the chunks are processed in their original order.

This pipeline allows for detailed analysis of customer frustration in long support transcripts, which would be challenging to process in a single pass due to token limitations or degraded LLM performance on very long inputs.

Best Practices

1. **Choose the Right Splitting Method**: Use the token count method when working with models that have strict token limits. Use the delimiter method when you need to split at logical boundaries in your text.
2. **Balance Chunk Size**: When using the token count method, choose a chunk size that balances between context preservation and model performance. Smaller chunks may lose context, while larger chunks may degrade model performance. The DocETL optimizer can find the chunk size that works best for your task, if you choose to use the optimizer.
3. **Consider Overlap**: In some cases, you might want to implement overlap between chunks to maintain context. This isn't built into the Split operation, but you can achieve it by post-processing the split chunks.
4. **Use Appropriate Delimiters**: When using the delimiter method, choose a delimiter that logically divides your text. Common choices include double newlines for

paragraphs, or custom markers for document sections. When using the delimiter method, adjust the `num_splits_to_group` parameter to create chunks that contain an appropriate amount of context for your task.

5. **Mind the Order**: If the order of chunks matters for your analysis, always set `associative: false` in your subsequent reduce operations.
6. **Optimize for Performance**: For very large documents, consider using a combination of delimiter and token count methods. First split into large sections using delimiters, then apply token count splitting to ensure no chunk exceeds model limits.

By leveraging the Split operation effectively, you can process large documents efficiently and extract meaningful insights using subsequent map and reduce operations.

Gather Operation

The Gather operation in DocETL is designed to maintain context when processing divided documents. It complements the Split operation by adding contextual information from surrounding chunks to each segment.

Motivation

When splitting long documents, such as complex legal contracts or court transcripts, individual chunks often lack sufficient context for accurate analysis or processing. This can lead to several challenges:

- Loss of reference information (e.g., terms defined in earlier sections)
- Incomplete representation of complex clauses that span multiple chunks
- Difficulty in understanding the broader document structure
- Missing important context from preambles or introductory sections

Context Challenge in Legal Documents

Imagine a lengthy merger agreement split into chunks. A single segment might contain clauses referencing "the Company" or "Effective Date" without clearly defining these terms. Without context from previous chunks, it becomes challenging to interpret the legal implications accurately.

How Gather Works

The Gather operation addresses these challenges by:

1. Identifying relevant surrounding chunks (peripheral context)
2. Adding this context to each chunk
3. Preserving document structure information

Peripheral Context

Peripheral context refers to the surrounding text or information that helps provide a more complete understanding of a specific chunk of content. In legal documents, this can

include:

- Preceding text that introduces key terms, parties, or conditions
- Following text that elaborates on clauses presented in the current chunk
- Document structure information, such as article or section headers
- Summarized versions of nearby chunks for efficient context provision

Document Structure

The Gather operation can maintain document structure through header hierarchies. This is particularly useful for preserving the overall structure of complex legal documents like contracts, agreements, or regulatory filings.

🚀 Example: Enhancing Context in Legal Document Analysis

Let's walk through an example of using the Gather operation to process a long merger agreement.

Step 1: Extract Metadata (Map operation before splitting)

First, we extract important metadata from the full document:

```yaml
- name: extract_metadata
  type: map
  prompt: |
    Extract the following metadata from the merger agreement:
    1. Agreement Date
    2. Parties involved
    3. Total value of the merger (if specified)

    Agreement text:
    {{ input.agreement_text }}

    Return the extracted information in a structured format.
  output:
    schema:
      agreement_date: string
      parties: list[string]
      merger_value: string
```

Step 2: Split Operation

Next, we split the document into manageable chunks:

```yaml
- name: split_merger_agreement
  type: split
  split_key: agreement_text
  method: token_count
  method_kwargs:
    token_count: 1000
```

Step 3: Extract Headers (Map operation)

We extract headers from each chunk:

```yaml
- name: extract_headers
  type: map
  input:
    - agreement_text_chunk
  prompt: |
    Extract any section headers from the following merger agreement chunk:
    {{ input.agreement_text_chunk }}
    Return the headers as a list, preserving their hierarchy.
  output:
    schema:
      headers: "list[{header: string, level: integer}]"
```

Step 4: Gather Operation

Now, we apply the Gather operation:

```yaml
- name: context_gatherer
  type: gather
  content_key: agreement_text_chunk
  doc_id_key: split_merger_agreement_id
  order_key: split_merger_agreement_chunk_num
  peripheral_chunks:
    previous:
      middle:
        content_key: agreement_text_chunk_summary
      tail:
        content_key: agreement_text_chunk
    next:
      head:
        count: 1
        content_key: agreement_text_chunk
  doc_header_key: headers
```

Step 5: Analyze Chunks (Map operation after Gather)

Finally, we analyze each chunk with its gathered context:

```yaml
- name: analyze_chunks
  type: map
```

```
  input:
    - agreement_text_chunk_rendered
    - agreement_date
    - parties
    - merger_value
  prompt: |
    Analyze the following chunk of a merger agreement, considering the
provided metadata:

    Agreement Date: {{ input.agreement_date }}
    Parties: {{ input.parties | join(', ') }}
    Merger Value: {{ input.merger_value }}

    Chunk content:
    {{ input.agreement_text_chunk_rendered }}

    Provide a summary of key points and any potential legal implications in
this chunk.
  output:
    schema:
      summary: string
      legal_implications: list[string]
```

This configuration:

1. Extracts important metadata from the full document before splitting
2. Splits the document into manageable chunks
3. Extracts headers from each chunk
4. Gathers context for each chunk, including:
5. Summaries of the chunks before the previous chunk
6. The full content of the previous chunk
7. The full content of the current chunk
8. The full content of the next chunk
9. Extracted headers for levels directly above headers in the current chunk, for structural context
10. Analyzes each chunk with its gathered context and the extracted metadata

Configuration

The Gather operation includes several key components:

- `type`: Always set to "gather"
- `doc_id_key`: Identifies chunks from the same original document
- `order_key`: Specifies the sequence of chunks within a group

- `content_key`: Indicates the field containing the chunk content
- `peripheral_chunks`: Specifies how to include context from surrounding chunks
- `doc_header_key` (optional): Denotes a field representing extracted headers for each chunk
- `sample` (optional): Number of samples to use for the operation

Peripheral Chunks Configuration

The `peripheral_chunks` configuration in the Gather operation is highly flexible, allowing users to precisely control how context is added to each chunk. This configuration determines which surrounding chunks are included and how they are presented.

Structure

The `peripheral_chunks` configuration is divided into two main sections:

1. `previous`: Defines how chunks preceding the current chunk are included.
2. `next`: Defines how chunks following the current chunk are included.

Each of these sections can contain up to three subsections:

- `head`: The first chunk(s) in the section.
- `middle`: Chunks between the `head` and `tail` sections.
- `tail`: The last chunk(s) in the section.

Configuration Options

For each subsection, you can specify:

- `count`: The number of chunks to include (for `head` and `tail` only).
- `content_key`: The key in the chunk data that contains the content to use.

Example Configuration

```yaml
peripheral_chunks:
  previous:
    head:
      count: 1
      content_key: full_content
    middle:
      content_key: summary_content
    tail:
      count: 2
      content_key: full_content
  next:
    head:
```

```
        count: 1
        content_key: full_content
```

This configuration would:

1. Include the full content of the very first chunk.
2. Include summaries of all chunks between the `head` and `tail` of the previous section.
3. Include the full content of 2 chunks immediately before the current chunk.
4. Include the full content of 1 chunk immediately after the current chunk.

Behavior Details

1. **Content Selection**: If a `content_key` is specified that's different from the main content key, it's treated as a summary. This is useful for including condensed versions of chunks in the `middle` section to save space. If no `content_key` is specified, it defaults to the main content key of the operation.
2. **Chunk Ordering**: For the `previous` section, chunks are processed in reverse order (from the current chunk towards the beginning of the document). For the `next` section, chunks are processed in forward order.
3. **Skipped Content**: If there are gaps between included chunks, the operation inserts a note indicating how many characters were skipped. Example: `[... 5000 characters skipped ...]`
4. **Chunk Labeling**: Each included chunk is labeled with its order number and whether it's a summary. Example: `[Chunk 5 (Summary)]` or `[Chunk 6]`

Best Practices

1. **Balance Context and Conciseness**: Use full content for immediate context (`head`) and summaries for `middle` sections to provide context without overwhelming the main content.
2. **Adapt to Document Structure**: Adjust the `count` for `head` and `tail` based on the typical length of your document sections.
3. **Use Asymmetric Configurations**: You might want more previous context than next context, or vice versa, depending on your specific use case.
4. **Consider Performance**: Including too much context can increase processing time and token usage. Use summaries and selective inclusion to optimize performance.

By leveraging this flexible configuration, you can tailor the Gather operation to provide the most relevant context for your specific document processing needs, balancing

completeness with efficiency.

Output

The Gather operation adds a new field to each input document, named by appending "_rendered" to the `content_key`. This field contains:

1. The reconstructed header hierarchy (if applicable)
2. Previous context (if any)
3. The main chunk, clearly marked
4. Next context (if any)
5. Indications of skipped content between contexts

Sample Output for Merger Agreement

```
agreement_text_chunk_rendered: |
  _Current Section:_ # Article 5: Representations and Warranties > ## 5.1
Representations and Warranties of the Company

  --- Previous Context ---
  [Chunk 17] ... summary of earlier sections on definitions and parties ...
  [... 500 characters skipped ...]
  [Chunk 18] The Company hereby represents and warrants to Parent and Merger
Sub as follows, except as set forth in the Company Disclosure Schedule:
  --- End Previous Context ---

  --- Begin Main Chunk ---
  5.1.1 Organization and Qualification. The Company is duly organized, validly
existing, and in good standing under the laws of its jurisdiction of
organization and has all requisite corporate power and authority to own, lease,
and operate its properties and to carry on its business as it is now being
conducted...
  --- End Main Chunk ---

  --- Next Context ---
  [Chunk 20] 5.1.2 Authority Relative to This Agreement. The Company has all
necessary corporate power and authority to execute and deliver this Agreement,
to perform its obligations hereunder, and to consummate the Merger...
  [... 750 characters skipped ...]
  --- End Next Context ---
```

Handling Document Structure

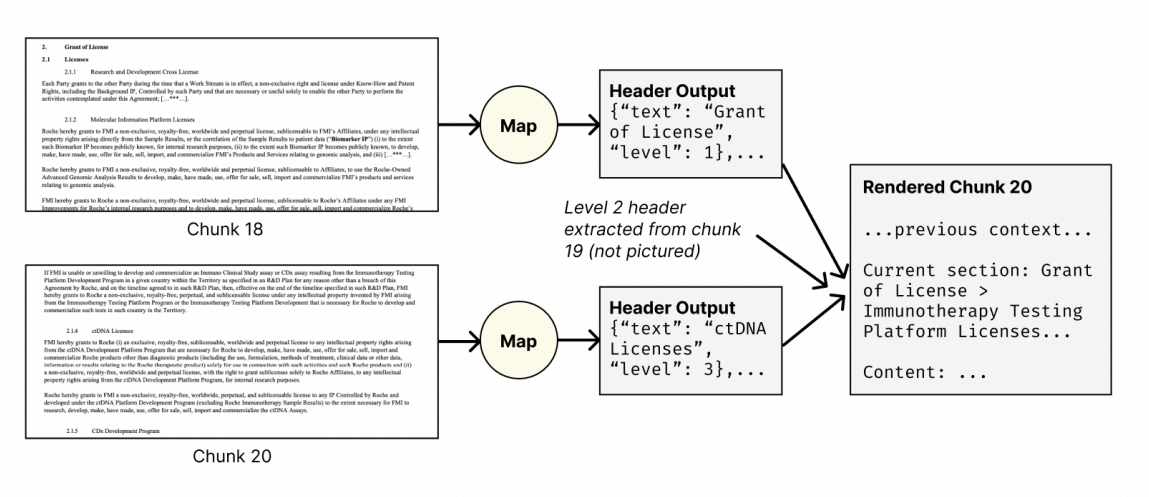
A key feature of the Gather operation is its ability to maintain document structure through header hierarchies. This is particularly useful for preserving the overall structure of complex documents like legal contracts, technical manuals, or research papers.

How Header Handling Works

1. Headers are typically extracted from each chunk using a Map operation after the Split but before the Gather.
2. The Gather operation uses these extracted headers to reconstruct the relevant header hierarchy for each chunk.
3. When rendering a chunk, the operation includes all the most recent headers from higher levels found in previous chunks.
4. This ensures that each rendered chunk includes a complete "path" of headers leading to its content, preserving the document's overall structure and context.

Example: Header Handling in Legal Contracts

Let's look at an example of how the Gather operation handles document headers in the context of legal contracts:



In this figure:

1. We see two chunks (18 and 20) from a 74-page legal contract.
2. Each chunk goes through a Map operation to extract headers.
3. For Chunk 18, a level 1 header "Grant of License" is extracted.
4. For Chunk 20, a level 3 header "ctDNA Licenses" is extracted.
5. When rendering Chunk 20, the Gather operation includes:
6. The level 1 header from Chunk 18 ("Grant of License")
7. A level 2 header from Chunk 19 (not pictured, but included in the rendered output)
8. The current level 3 header from Chunk 20 ("ctDNA Licenses")

This hierarchical structure provides crucial context for understanding the content of Chunk 20, even though the higher-level headers are not directly present in that chunk.

> **Importance of Header Hierarchy**
>
> By maintaining the header hierarchy, the Gather operation ensures that each chunk is placed in its proper context within the overall document structure. This is especially crucial for complex documents where understanding the relationship between different sections is key to accurate analysis or processing.

Best Practices

1. **Extract Metadata Before Splitting**: Run a map operation on the full document before splitting to extract any metadata that could be useful when processing any chunk (e.g., agreement dates, parties involved). Reference this metadata in subsequent map operations after the gather step.
2. **Balance Context and Efficiency**: For ultra-long documents, consider using summarized versions of chunks in the "middle" sections to strike a balance between providing context and managing the overall size of the processed data.
3. **Preserve Document Structure**: Utilize the `doc_header_key` parameter to include relevant structural information from the original document, which can be important for understanding the context of complex or structured content.
4. **Tailor Context to Your Task**: Adjust the `peripheral_chunks` configuration based on the specific needs of your analysis. Some tasks may require more preceding context, while others might benefit from more following context.
5. **Combine with Other Operations**: The Gather operation is most powerful when used in conjunction with Split, Map (for summarization or header extraction), and subsequent analysis operations to process long documents effectively.
6. **Consider Performance**: Be mindful of the increased token count when adding context. Adjust your downstream operations accordingly to handle the larger input sizes, and use summarized context where appropriate to manage token usage.
7. **Use `next` Sparingly**: The `next` parameter in the Gather operation should be used judiciously. It's primarily beneficial in specific scenarios:
   - When dealing with structured data or tables where the next chunk provides essential context for understanding the current chunk.
   - In cases where the end of a chunk might cut off a sentence or important piece of information that continues in the next chunk.

For most text-based documents, focusing on the `prev` context is usually sufficient. Overuse of `next` can lead to unnecessary token consumption and potential redundancy in the gathered output.

When to Use `next`

Consider using `next` when processing:

```
- Financial reports with multi-page tables
- Technical documents where diagrams span multiple pages
- Legal contracts where clauses might be split across chunk boundaries
```

By default, it's recommended to set `next=0` unless you have a specific need for forward context in your document processing pipeline.

Unnest Operation

The Unnest operation in DocETL is designed to expand an array field or a dictionary in the input data into multiple items. This operation is particularly useful when you need to process or analyze individual elements of an array or specific fields of a nested dictionary separately.

How Unnest Works

The Unnest operation behaves differently depending on the type of data being unnested:

- For list-type unnesting: It replaces the original key with each individual element from the list.
- For dictionary-type unnesting: It adds new keys to the parent dictionary based on the `expand_fields` parameter.

Unnest does not have an output schema. It modifies the structure of your data in place.

Motivation

The Unnest operation is valuable in scenarios where you need to:

- Process individual items from a list of products in an order
- Analyze separate entries in a list of comments or reviews
- Expand nested data structures for more granular processing
- Flatten complex data structures for easier analysis

Configuration

Required Parameters

Parameter	Description
type	Must be set to "unnest"
name	A unique name for the operation

Parameter	Description
unnest_key	The key of the array field to unnest

Optional Parameters

Parameter	Description	Default
keep_empty	If true, empty arrays being exploded will be kept in the output (with value None)	false
expand_fields	A list of fields to expand from the nested dictionary into the parent dictionary, if unnesting a dict	[]
recursive	If true, the unnest operation will be applied recursively to nested arrays	false
depth	The maximum depth for recursive unnesting (only applicable if recursive is true)	inf
sample	Number of samples to use for the operation	None

Output

The Unnest operation modifies the structure of your data:

- For list-type unnesting: It generates multiple output items for each input item, replacing the original array in the `unnest_key` field with individual elements.
- For dictionary-type unnesting: It expands the specified fields into the parent dictionary.

All other original key-value pairs from the input item are preserved in the output.

Note

When unnesting dictionaries, the original nested dictionary is preserved in the output, and the specified fields are expanded into the parent dictionary.

Use Cases

1. **Product Analysis in Orders**: Unnest a list of products in each order, then use a map operation to analyze each product individually.
2. **Comment Sentiment Analysis**: Unnest a list of comments for each post, enabling sentiment analysis on individual comments.
3. **Nested Data Structure Flattening**: Unnest complex nested data structures to create a flattened dataset for easier analysis or processing.
4. **Processing Time Series Data**: Unnest time series data stored in arrays to analyze individual time points.

Example: Analyzing Product Reviews

Let's walk through an example of using the Unnest operation to prepare product reviews for detailed analysis.

```
- name: extract_salient_quotes
  type: map
  prompt: |
    For the following product review, extract up to 3 salient quotes that best
represent the reviewer's opinion:

    {{ input.review_text }}

    For each quote, provide the text and its sentiment (positive, negative, or
neutral).
  output:
    schema:
      salient_quotes: list[string]

- name: unnest_quotes
  type: unnest
  unnest_key: salient_quotes

- name: analyze_quote
  type: map
  prompt: |
    Analyze the following quote from a product review:

    Quote & information: {{ input.salient_quotes }}
    Review text: {{ input.review_text }}

    Provide a detailed analysis of the quote, including:
    1. The specific aspect of the product being discussed
    2. The strength of the sentiment (-5 to 5, where -5 is extremely negative
and 5 is extremely positive)
    3. Any key terms or phrases that stand out

  output:
    schema:
```

```
        product_aspect: string
        sentiment_strength: number
        key_terms: list[string]
```

This example demonstrates how the Unnest operation fits into a pipeline for analyzing product reviews:

1. The first Map operation extracts salient quotes from each review.
2. The Unnest operation expands the 'salient_quotes' array, creating individual items for each quote. Each quote can now be accessed via `input.salient_quotes`.
3. The second Map operation performs a detailed analysis on each individual quote.

By unnesting the quotes, we enable more granular analysis that wouldn't be possible if we processed the entire review as a single unit.

Advanced Features

Recursive Unnesting

When dealing with deeply nested structures, you can use the `recursive` parameter to apply the unnest operation at multiple levels:

```
- name: recursive_unnest
  type: unnest
  unnest_key: nested_data
  recursive: true
  depth: 3 # Limit recursion to 3 levels deep
```

Dictionary Expansion

When unnesting dictionaries, you can use the `expand_fields` parameter to flatten specific fields into the parent structure:

```
- name: expand_user_data
  type: unnest
  unnest_key: user_info
  expand_fields:
    - name
    - age
    - location
```

In this case, `name`, `age`, and `location` would be added as new keys in the parent dictionary, alongside the original `user_info` key.

Best Practices

1. **Choose the Right Unnest Key**: Ensure you're unnesting the correct field that contains the array or nested structure you want to expand.
2. **Consider Data Volume**: Unnesting can significantly increase the number of items in your data stream. Be mindful of this when designing subsequent operations in your pipeline.
3. **Use Expand Fields Wisely**: When unnesting dictionaries, use the `expand_fields` parameter to flatten your data structure if needed, but be cautious of potential key conflicts.
4. **Handle Empty Arrays**: Decide whether empty arrays should be kept (using `keep_empty`) based on your specific use case and how subsequent operations should handle null values.
5. **Preserve Context**: When unnesting, consider whether you need to carry forward any context from the parent item. The unnest operation preserves all other fields, which helps maintain context.

Sample operation

The Sample operation in DocETL samples items from the input. It is meant mostly as a debugging tool:

Insert it before the last operation, the one you're currently trying to add to the end of a working pipeline, to limit the amount of data it will be fed, so that the run time is small enough to comfortably debug its prompt. Once it seems to be working, you can remove the sample operation. You can then repeat this for each operation you add while developing your pipeline!

🚀 Example:

```yaml
- name: sample_concepts
  type: sample
  method: uniform
  samples: 0.1
  stratify_key: category
  random_state: 42
```

This sample operation will return a pseudo-randomly selected 10% of the samples (samples: 0.1). The random selection will be seeded with a constant (42), meaning the same sample will be returned if you rerun the pipeline (If no random state is given, a different sample will be returned every time). Additionally, the random sampling will sample each value of the category key proportionally.

Required Parameters

- name: A unique name for the operation.
- type: Must be set to "sample".
- method: The sampling method to use. Can be "uniform", "outliers", "custom", "first", "top_embedding", or "top_fts".
- samples: Either a list of key-value pairs representing document ids and values, an integer count of samples, or a float fraction of samples.

Optional Parameters

Parameter	Description	Default
random_state	An integer to seed the random generator with	None
stratify_key	Key(s) to stratify by. Can be a string or list of strings	None
samples_per_group	When stratifying, sample N items per group vs. proportionally	False
method_kwargs	Additional parameters for specific methods (e.g., outliers)	{}

Sampling Methods

Uniform Sampling

Randomly samples items from the input data. When combined with stratification, maintains the distribution of the stratified groups.

```yaml
- name: uniform_sample
  type: sample
  method: uniform
  samples: 100
```

First Sampling

Takes the first N items from the input. When combined with stratification, takes proportionally from each group.

```yaml
- name: first_sample
  type: sample
  method: first
  samples: 50
```

Outlier Sampling

Samples based on distance from a center point in embedding space. Specify the following in method_kwargs:

- embedding_keys: A list of keys to use for creating embeddings.
- std: The number of standard deviations to use as the cutoff for outliers.

- samples: The number or fraction of samples to consider as outliers.
- keep: Whether to keep (true) or remove (false) the outliers. Defaults to false.
- center: (Optional) A dictionary specifying the center point for distance calculations.

You must specify either "std" or "samples" in the method_kwargs, but not both.

```yaml
- name: remove_outliers
  type: sample
  method: outliers
  method_kwargs:
    embedding_keys:
      - concept
      - description
    std: 2
    keep: false
```

Custom Sampling

Samples specific items by matching key-value pairs. Stratification is not supported with custom sampling.

```yaml
- name: custom_sample
  type: sample
  method: custom
  samples:
    - id: 1
    - id: 5
```

Top Embedding Sampling

Retrieves the top N most similar items to a query based on semantic similarity using embeddings. Requires the following in method_kwargs:

- keys: A list of keys to use for creating embeddings
- query: The query string to match against (supports Jinja templates)
- embedding_model: (Optional) The embedding model to use. Defaults to "text-embedding-3-small"

```yaml
- name: semantic_search
  type: sample
  method: top_embedding
  samples: 10
  method_kwargs:
    keys:
      - title
      - content
```

```
    query: "machine learning applications in healthcare"
    embedding_model: text-embedding-3-small
```

With Jinja template for dynamic queries:

```
- name: personalized_search
  type: sample
  method: top_embedding
  samples: 5
  method_kwargs:
    keys:
      - description
    query: "{{ input.user_query }}"
```

Top FTS Sampling

Retrieves the top N items using full-text search with BM25 algorithm. Requires the following in method_kwargs:

- keys: A list of keys to search within
- query: The query string for keyword matching (supports Jinja templates)

```
- name: keyword_search
  type: sample
  method: top_fts
  samples: 20
  method_kwargs:
    keys:
      - title
      - content
      - tags
    query: "python programming tutorial"
```

With dynamic query:

```
- name: search_products
  type: sample
  method: top_fts
  samples: 0.1  # Top 10% of results
  method_kwargs:
    keys:
      - product_name
      - description
    query: "{{ input.search_terms }}"
```

Stratification

Stratification can be applied to "uniform", "first", "outliers", "top_embedding", and "top_fts" methods. It ensures that the sample maintains the distribution of specified key(s) in the data or retrieves top items from each stratum.

Single Key Stratification

```yaml
- name: stratified_sample
  type: sample
  method: uniform
  samples: 0.2
  stratify_key: category
```

Multiple Key Stratification

When using multiple keys, stratification is based on the combination of values:

```yaml
- name: multi_stratified_sample
  type: sample
  method: uniform
  samples: 50
  stratify_key:
    - type
    - size
```

Samples Per Group

Instead of proportional sampling, you can sample a fixed number from each stratum:

```yaml
- name: stratified_per_group
  type: sample
  method: uniform
  samples: 10  # Sample 10 items from each group
  stratify_key: category
  samples_per_group: true
```

This also works with fractions:

```yaml
- name: stratified_fraction_per_group
  type: sample
  method: uniform
  samples: 0.3  # Sample 30% from each group
  stratify_key: category
  samples_per_group: true
```

Complete Examples

Stratified outlier detection:

```yaml
- name: stratified_outliers
  type: sample
  method: outliers
  stratify_key: document_type
  method_kwargs:
    embedding_keys:
      - title
      - content
    std: 1.5
    keep: false
```

Stratified first sampling with multiple keys:

```yaml
- name: stratified_first
  type: sample
  method: first
  samples: 100
  stratify_key:
    - category
    - priority
  samples_per_group: false  # Take proportionally from each combination
```

Outlier sampling with a custom center:

```yaml
- name: centered_outliers
  type: sample
  method: outliers
  method_kwargs:
    embedding_keys:
      - concept
      - description
    center:
      concept: Tree house
      description: A small house built among the branches of a tree for
children to play in.
    samples: 20  # Keep the 20 furthest items from the center
    keep: true
```

Stratified semantic search - retrieve top documents from each category:

```yaml
- name: stratified_semantic_search
  type: sample
  method: top_embedding
  samples: 5  # Get top 5 from each category
  stratify_key: category
  samples_per_group: true
  method_kwargs:
    keys:
      - title
      - abstract
    query: "recent advances in artificial intelligence"
```

Full-text search with multiple stratification keys:

```yaml
- name: stratified_keyword_search
  type: sample
  method: top_fts
  samples: 3
  stratify_key:
    - department
    - priority
  samples_per_group: true
  method_kwargs:
    keys:
      - subject
      - content
    query: "urgent customer complaint refund"
```

Note on TopK Operation

For retrieval use cases, consider using the dedicated TopK operation which provides a cleaner interface specifically designed for top-k retrieval with three methods: - `embedding`: Semantic similarity search - `fts`: Full-text search using BM25 - `llm_compare`: LLM-based ranking

The TopK operation offers the same functionality as the sample operation's `top_embedding` and `top_fts` methods, but with a more intuitive API for retrieval tasks.

TopK Operation

The TopK operation retrieves the most relevant items from your dataset using three different retrieval methods: semantic similarity, full-text search, or LLM-based comparison. It provides a clean, specialized interface for retrieval tasks where you need to find and rank the best matching documents based on specific criteria.

Overview

TopK is designed for retrieval and ranking use cases such as finding relevant documents for a query, filtering large datasets to the most important items, implementing retrieval-augmented generation (RAG) pipelines, and building recommendation systems. Unlike general sampling operations, TopK focuses specifically on retrieving the best matches according to your chosen method and criteria.

The operation supports three distinct retrieval methods, each optimized for different use cases. The **embedding method** uses semantic similarity to find conceptually related documents, making it ideal when meaning matters more than exact words. The **full-text search (FTS) method** employs the BM25 algorithm for keyword-based retrieval, perfect for when specific terms are important. The **LLM compare method** leverages a language model to rank documents based on complex criteria that require reasoning and multi-factor comparisons.

Configuration

Core Parameters

Parameter	Type	Description	Required
`method`	`"embedding"` | `"fts"` | `"llm_compare"`	Retrieval method to use	Yes
`k`	`int` or `float`	Number of items to retrieve (float = percentage)	Yes

Parameter	Type	Description	Required
`keys`	`list[str]`	Document fields to use for matching/comparison	Yes
`query`	`str`	Query or ranking criteria (Jinja templates supported for `embedding` and `fts` only)	Yes

Method-Specific Parameters

Parameter	Type	Methods	Description	Default
`embedding_model`	`str`	`embedding`, `llm_compare`	Model for embeddings	`"text-embedding-3-small"`
`model`	`str`	`llm_compare`	LLM model for comparisons	Required for `llm_compare`
`batch_size`	`int`	`llm_compare`	Batch size for LLM ranking	`10`
`stratify_key`	`str` or `list[str]`	`embedding`, `fts`	Keys for stratified retrieval	`None`

Examples

Semantic Search with Embeddings

When you need to find documents based on meaning rather than exact keywords, the embedding method excels. This example finds support tickets semantically similar to payment processing issues:

```yaml
- name: find_relevant_tickets
  type: topk
  method: embedding
  k: 5
  keys:
    - subject
```

```
    - description
    - customer_feedback
  query: "payment processing errors with international transactions"
  embedding_model: text-embedding-3-small
```

Keyword Search with FTS

For cases where specific terms matter, such as searching a product catalog, the FTS method provides fast, accurate keyword matching without API costs:

```
- name: search_products
  type: topk
  method: fts
  k: 20
  keys:
    - product_name
    - description
    - category
    - tags
  query: "wireless noise cancelling headphones bluetooth"
```

Complex Ranking with LLM Compare

When you need to rank items based on multiple factors or subjective criteria, the LLM compare method allows you to specify complex ranking logic. Note that this method requires consistent criteria across all documents and doesn't support Jinja templates:

```
- name: screen_resumes
  type: topk
  method: llm_compare
  k: 10
  keys:
    - skills
    - experience
    - education
  query: |
    Rank candidates based on their fit for a Senior Backend Engineer role
requiring:
    - 5+ years Python experience
    - Distributed systems expertise
    - Strong knowledge of PostgreSQL and Redis
    - Experience with microservices architecture
    - Leadership experience is a plus

    Prioritize hands-on technical experience over academic credentials.
  model: gpt-4o
  batch_size: 5
```

Dynamic Queries with Templates

The embedding and FTS methods support Jinja templates for dynamic queries that adapt based on input data. This enables personalized search experiences:

```yaml
- name: personalized_search
  type: topk
  method: embedding
  k: 10
  keys:
    - content
    - tags
  query: |
    {{ input.user_preferences }}
    Focus on {{ input.topic_of_interest }}
    Exclude anything related to {{ input.blocked_topics }}
```

Stratified Retrieval

Both embedding and FTS methods support stratification, which ensures you retrieve top items from each group. This is useful for maintaining diversity in results:

```yaml
- name: recommendations_by_category
  type: topk
  method: fts
  k: 3  # Get top 3 from each category
  keys:
    - product_name
    - description
  query: "premium quality bestseller"
  stratify_key: category
```

Common Patterns

Single-Document RAG Pipeline

A retrieval-augmented generation pipeline for answering questions about a single document typically retrieves the most relevant chunks, then synthesizes them into a coherent answer using reduce:

```yaml
# Step 1: Retrieve most relevant document chunks
- name: retrieve_context
  type: topk
  method: embedding
  k: 5
  keys: [content]
  query: "{{ input.user_question }}"

# Step 2: Generate comprehensive answer from all retrieved chunks
- name: generate_answer
  type: reduce
```

```
  reduce_key: user_question  # Group by the question
  prompt: |
    Based on the following document excerpts, provide a comprehensive answer
to the question.

    Question: {{ inputs[0].user_question }}

    Retrieved context from document:
    {% for chunk in inputs %}
    - {{ chunk.content }}
    {% endfor %}

    Synthesize the information from all excerpts into a single, coherent
answer.
  output_schema:
    answer: string
```

Multi-Stage Filtering

For complex retrieval tasks, you might combine multiple TopK operations with different methods, progressively refining your results:

```
# Cast a wide net with keyword search
- name: initial_search
  type: topk
  method: fts
  k: 100
  keys: [title, content]
  query: "machine learning"

# Refine with semantic search
- name: refine_results
  type: topk
  method: embedding
  k: 20
  keys: [title, content]
  query: "practical applications of deep learning in healthcare"

# Final ranking with LLM
- name: final_ranking
  type: topk
  method: llm_compare
  k: 5
  keys: [title, abstract, impact_factor]
  query: "Rank by potential clinical impact and implementation feasibility"
  model: gpt-4o
```

Performance Considerations

Each retrieval method has different performance characteristics that should guide your choice. The **FTS method** is the fastest and has no API costs since it uses local BM25

scoring, making it ideal for high-volume or cost-sensitive applications. The **embedding method** requires API calls to generate embeddings for both documents and queries, but once computed, similarity matching is very fast. It provides excellent semantic understanding but at a moderate cost. The **LLM compare method** has the highest cost and slowest performance due to multiple LLM calls, but offers unmatched flexibility for complex ranking criteria.

When optimizing performance, consider preprocessing embeddings offline for the embedding method, using FTS for initial filtering before applying more expensive methods, and carefully tuning the batch_size parameter for LLM compare to balance speed and accuracy.

Implementation Details

Embedding Method

The embedding method converts both your documents and query into high-dimensional vectors using the specified embedding model (defaulting to OpenAI's text-embedding-3-small). It then calculates cosine similarity between the query vector and each document vector, returning the k documents with highest similarity scores. The method handles text normalization and truncation automatically, and when stratification is enabled, it performs this process independently for each stratum.

FTS Method

The full-text search method uses the BM25 algorithm, the same ranking function used by search engines like Elasticsearch. Documents are tokenized and lowercase-normalized, with special characters removed. The BM25 scoring function considers term frequency (with saturation to prevent overweighting repeated terms), inverse document frequency (giving more weight to rare terms), and document length normalization. The implementation uses the rank-bm25 library for efficient scoring.

LLM Compare Method

The LLM compare method delegates to the rank operation, which implements sophisticated comparison algorithms from human-powered sorting research. It starts with an initial ordering using embeddings, then applies sliding windows of documents for pairwise comparison by the LLM. The LLM evaluates documents in batches (controlled by batch_size) and produces rankings based on the specified criteria. This method requires consistent ranking criteria across all documents, which is why Jinja templates are not supported—the LLM needs to compare all documents using the same criteria for fair ranking.

Error Handling

The TopK operation is designed to handle edge cases gracefully. If k exceeds the number of available documents, it returns all available items rather than failing. When specified keys are missing from some documents, it uses whatever fields are available. For the FTS method, documents with empty text after normalization are handled appropriately, and for the embedding method, API failures include automatic retries with exponential backoff.

Code Operations

Code operations in DocETL allow you to define transformations using Python code rather than LLM prompts. This is useful when you need deterministic processing, complex calculations, or want to leverage existing Python libraries.

Motivation

While LLM-powered operations are powerful for natural language tasks, sometimes you need operations that are:

- Deterministic and reproducible
- Integrated with external Python libraries
- Focused on structured data transformations
- Math-based or computationally intensive (something an LLM is not good at)

Code operations provide a way to handle these cases efficiently without LLM overhead.

Types of Code Operations

Code Map Operation

The Code Map operation applies a Python function to each item in your input data independently.

Example Code Map Operation

```yaml
- name: extract_keywords
  type: code_map
  code: |
    def transform(doc) -> dict:
        # Your transformation code here
        keywords = doc['text'].lower().split()
        return {
            'keywords': keywords,
            'keyword_count': len(keywords)
        }
```

The code must define a `transform` function that takes a single document as input and returns a dictionary of transformed values.

Code Reduce Operation

The Code Reduce operation aggregates multiple items into a single result using a Python function.

Example Code Reduce Operation

```yaml
- name: aggregate_stats
  type: code_reduce
  reduce_key: category
  code: |
    def transform(items) -> dict:
        total = sum(item['value'] for item in items)
        avg = total / len(items)
        return {
            'total': total,
            'average': avg,
            'count': len(items)
        }
```

The transform function for reduce operations takes a list of items as input and returns a single aggregated result.

Code Filter Operation

The Code Filter operation allows you to filter items based on custom Python logic.

Example Code Filter Operation

```yaml
- name: filter_valid_entries
  type: code_filter
  code: |
    def transform(doc) -> bool:
        # Return True to keep the document, False to filter it out
        return doc['score'] >= 0.5 and len(doc['text']) > 100
```

The transform function should return True for items to keep and False for items to filter out.

Configuration

Required Parameters

- type: Must be "code_map", "code_reduce", or "code_filter"
- code: Python code containing the transform function. For map, the function must take a single document as input and return a document (a dictionary). For reduce, the function must take a list of documents as input and return a single aggregated document (a dictionary). For filter, the function must take a single document as input and return a boolean value indicating whether to keep the document.

Optional Parameters

Parameter	Description	Default
drop_keys	List of keys to remove from output (code_map only)	None
reduce_key	Key(s) to group by (code_reduce only)	"_all"
pass_through	Pass through unmodified keys from first item in group (code_reduce only)	false
concurrent_thread_count	The number of threads to start	the number of logical CPU cores (os.cpu_count())

Python API

The DocETL Python API provides a programmatic way to define, optimize, and run document processing pipelines. This approach offers an alternative to the YAML configuration method, allowing for more dynamic and flexible pipeline construction.

Overview

The Python API consists of several classes:

- Dataset: Represents a dataset with a type and path.
- Various operation classes (e.g., MapOp, ReduceOp, FilterOp) for different types of data processing steps.
- PipelineStep: Represents a step in the pipeline with input and operations.
- Pipeline: The main class for defining and running a complete document processing pipeline.
- PipelineOutput: Defines the output configuration for the pipeline.

Example Usage

Here's an example of how to use the Python API to create and run a simple document processing pipeline:

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

# Define datasets
datasets = {
    "my_dataset": Dataset(type="file", path="input.json", parsing=
[{"input_key": "file_path", "function": "txt_to_string", "output_key":
"content"}]),
}

# Note that the parsing is applied to the `file_path` key in each item of the
dataset,
# and the result is stored in the `content` key.

# Define operations
operations = [
    MapOp(
        name="process",
        type="map",
```

```
        prompt="Determine what type of document this is: {{ input.content }}",
        output={"schema": {"document_type": "string"}}
    ),
    ReduceOp(
        name="summarize",
        type="reduce",
        reduce_key="document_type",
        prompt="Summarize the processed contents: {% for item in inputs %}{{
item.content }} {% endfor %}",
        output={"schema": {"summary": "string"}}
    )
]

# Define pipeline steps
steps = [
    PipelineStep(name="process_step", input="my_dataset", operations=
["process"]),
    PipelineStep(name="summarize_step", input="process_step", operations=
["summarize"])
]

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

# Create the pipeline
pipeline = Pipeline(
    name="example_pipeline",
    datasets=datasets,
    operations=operations,
    steps=steps,
    output=output,
    default_model="gpt-4o-mini"
)

# Optimize the pipeline
optimized_pipeline = pipeline.optimize()

# Run the optimized pipeline
result = optimized_pipeline.run() # Saves the result to the output path

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

This example demonstrates how to create a simple pipeline that processes input documents and then summarizes the processed content. The pipeline is optimized before execution to improve performance.

API Reference

For a complete reference of all available classes and their methods, please refer to the Python API Reference.

The API Reference provides detailed information about each class, including:

- Available parameters
- Method signatures
- Return types
- Usage examples

Python API Examples

This document provides two examples of how to use DocETL's Python API. Each example demonstrates a single pipeline step with multiple operations.

Example 1: Extract and Summarize Product Review Themes

This example extracts themes and quotes from product reviews, then summarizes the top themes across ALL documents using the special `_all` reduce key:

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

# Define dataset - A CSV of product reviews
dataset = Dataset(
    type="file",
    path="product_reviews.csv"  # Contains columns: review_id, product_id,
rating, review_text
)

# Define operations
operations = [
    # Extract themes and quotes from each review
    MapOp(
        name="extract_themes",
        type="map",
        prompt="""
        Analyze this product review and extract the key themes and
representative quotes:

        Review: {{ input.review_text }}
        Rating: {{ input.rating }}

        Identify 2-3 major themes (e.g., usability, quality, value) and
extract direct quotes that best represent each theme.
        """,
        output={
            "schema": {
                "themes": "list[string]",
                "quotes": "list[string]",
                "sentiment": "string"
            }
        }
    ),
    # Summarize all themes across reviews using _all key
    ReduceOp(
```

```
        name="summarize_themes",
        type="reduce",
        reduce_key="_all",  # Special key to reduce across all items
        prompt="""
        Analyze and synthesize the themes and quotes from these product
reviews:

        {% for item in inputs %}
        Review ID: {{ item.review_id }}
        Product: {{ item.product_id }}
        Rating: {{ item.rating }}
        Themes: {{ item.themes | join(", ") }}
        Quotes:
        {% for quote in item.quotes %}
        - "{{ quote }}"
        {% endfor %}
        Sentiment: {{ item.sentiment }}

        {% endfor %}

        Summarize the most frequent themes, the most representative quotes for
each theme, and the overall sentiment.
        """,
        output={
            "schema": {
                "summary": "string"
            }
        }
    )
]

# Define pipeline step (can consist of multiple operations)
step = PipelineStep(
    name="review_analysis",
    input="product_reviews",
    operations=["extract_themes", "summarize_themes"]
)

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

# Create and run pipeline
pipeline = Pipeline(
    name="review_analysis_pipeline",
    datasets={"product_reviews": dataset},
    operations=operations,
    steps=[step],
    output=output,
    default_model="gpt-4o-mini"
)

# Run the pipeline
cost = pipeline.run()
print(f"Pipeline execution completed. Total cost: ${cost:.2f}")
```

Example 2: Map-Unnest-Resolve-Reduce on Theme Keys

This example extracts theme-quote pairs from reviews, unnests them, resolves similar themes, and then reduces on the theme key. Here we will also optimize the pipeline.

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

# Define dataset - A JSON file with product reviews
dataset = Dataset(
    type="file",
    path="product_reviews.csv" # Same csv as previously
)

# Define operations
operations = [
    # Extract theme-quote pairs from each review
    MapOp(
        name="extract_theme_quotes",
        type="map",
        prompt="""
        Extract theme and quote pairs from this product review:

        Review: {{ input.review_text }}
        Product: {{ input.product_name }}
        Rating: {{ input.rating }}

        For each distinct theme in the review, extract a direct quote that
best represents that theme.
        Return each theme and its representative quote as a separate object in
the "theme_quotes" array.
        """,
        output={
            "schema": {
                "theme_quotes": "array"  # Array of objects with theme and
quote properties
            }
        }
    ),
    # Unnest to create separate items for each theme-quote pair
    UnnestOp(
        name="unnest_theme_quotes",
        type="unnest",
        array_path="theme_quotes"
    ),
    # Resolve similar themes using fuzzy matching
    ResolveOp(
        name="resolve_themes",
        type="resolve",
        comparison_prompt="""
        Determine if these two themes are the same or closely related:

        Theme 1: {{ input1.theme }}
        Theme 2: {{ input2.theme }}
```

```
        Consider semantic similarity, synonyms, and conceptual overlap.
        """,
        resolution_prompt="""
        Given the following list of similar themes, determine a canonical name
that best represents all of them:

        {% for item in inputs %}
        Theme: {{ item.theme }}
        {% endfor %}

        Choose a clear, concise name that accurately captures the core concept
shared across all these related themes.
        """
    ),
    # Reduce by theme to aggregate quotes and insights
    ReduceOp(
        name="aggregate_by_theme",
        type="reduce",
        reduce_key="theme",
        prompt="""
        Analyze all quotes related to the theme "{{ reduce_key }}":

        {% for item in inputs %}
        Product: {{ item.product_name }}
        Rating: {{ item.rating }}
        Quote: "{{ item.quote }}"

        {% endfor %}

        Summarize the key insights about this theme across all products and
ratings.
        """,
        output={
            "schema": {
                "summary": "string"
            }
        }
    )
]

# Define pipeline with a single step
step = PipelineStep(
    name="theme_analysis",
    input="product_reviews",
    operations=["extract_theme_quotes", "unnest_theme_quotes",
"resolve_themes", "aggregate_by_theme"]
)

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

# Create the pipeline
pipeline = Pipeline(
    name="theme_analysis_pipeline",
    datasets={"product_reviews": dataset},
    operations=operations,
    steps=[step],
```

```
    output=output,
    default_model="gpt-4o"
)

# Optimize the pipeline before running
optimized_pipeline = pipeline.optimize()

# Run the optimized pipeline
cost = optimized_pipeline.run()
print(f"Pipeline execution completed. Total cost: ${cost:.2f}")
```

Note that datasets can be json or CSV.

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 ↩↩

Semantic Operations

The pandas integration provides several semantic operations through the `.semantic` accessor. Each operation is designed to handle specific types of transformations and analyses using LLMs.

All semantic operations return a new DataFrame that preserves the original columns and adds new columns based on the output schema. For example, if your original DataFrame has a column `text` and you use `map` with an `output={"schema": {"sentiment": "str", "keywords": "list[str]"}}`, the resulting DataFrame will have three columns: `text`, `sentiment`, and `keywords`. This makes it easy to chain operations and maintain data lineage.

Map Operation

Apply semantic mapping to each row using a language model.

Documentation: https://ucbepic.github.io/docetl/operators/map/

Parameters:

Name	Type	Description	Default
`prompt`	`str`	Jinja template string for generating prompts. Use {{input.column_name}} to reference input columns.	*required*
`output`	`dict[str, Any]`	Dictionary containing output configuration with keys: - "schema": Dictionary defining the expected output structure and types. Example: {"entities": "list[str]", "sentiment": "str"} - "mode": Optional output mode. Either "tools" (default) or "structured_output". "structured_output" uses native JSON schema mode for supported models. - "n": Optional number of	`None`

Name	Type	Description	Default
		outputs to generate for each input (synthetic data generation)	
`output_schema`	`dict[str, Any]`	DEPRECATED. Use 'output' parameter instead. Dictionary defining the expected output structure for backward compatibility.	`None`
`**kwargs`		Additional configuration options: - model: LLM model to use (default: from config) - batch_prompt: Template for processing multiple documents in a single prompt - max_batch_size: Maximum number of documents to process in a single batch - optimize: Flag to enable operation optimization (default: True) - recursively_optimize: Flag to enable recursive optimization (default: False) - sample: Number of samples to use for the operation - tools: List of tool definitions for LLM use - validate: List of Python expressions to validate output - num_retries_on_validate_failure: Number of retry attempts (default: 0) - gleaning: Configuration for LLM-based refinement - drop_keys: List of keys to drop from input - timeout: Timeout for each LLM call in seconds (default: 120) - max_retries_per_timeout: Maximum retries per timeout (default: 2) - litellm_completion_kwargs: Additional parameters for LiteLLM - skip_on_error: Skip operation if LLM returns error (default: False) - bypass_cache: Bypass cache for this operation (default: False) - n: Number of outputs to generate for	`{}`

Name	Type	Description	Default
		each input (synthetic data generation)	

Returns:

Type	Description
`DataFrame`	pd.DataFrame: A new DataFrame containing the transformed data with columns matching the output schema.

Examples:

```
>>> # Extract entities and sentiment
>>> df.semantic.map(
...     prompt="Analyze this text: {{input.text}}",
...     output={
...         "schema": {
...             "entities": "list[str]",
...             "sentiment": "str"
...         }
...     },
...     validate=["len(output['entities']) <= 5"],
...     num_retries_on_validate_failure=2
... )
```

```
>>> # Generate synthetic data with multiple variations per input
>>> df.semantic.map(
...     prompt="Create a headline for: {{input.topic}}",
...     output={"schema": {"headline": "str"}, "n": 5}
... )
```

```
>>> # Use structured output mode for better JSON schema support
>>> df.semantic.map(
...     prompt="Extract structured data: {{input.text}}",
...     output={
...         "schema": {"name": "str", "age": "int", "tags": "list[str]"},
...         "mode": "structured_output"
...     }
... )
```

Source code in `docetl/apis/pd_accessors.py`

```
def map(
    self,
    prompt: str,
    output: dict[str, Any] = None,
    *,
    output_schema: dict[str, Any] = None,
    **kwargs,
) -> pd.DataFrame:
    """
    Apply semantic mapping to each row using a language model.

    Documentation: https://ucbepic.github.io/docetl/operators/map/

    Args:
        prompt: Jinja template string for generating prompts. Use
{{input.column_name}}
                to reference input columns.
        output: Dictionary containing output configuration with keys:
                - "schema": Dictionary defining the expected output
structure and types.
                            Example: {"entities": "list[str]", "sentiment":
"str"}
                - "mode": Optional output mode. Either "tools" (default)
or "structured_output".
                          "structured_output" uses native JSON schema mode
for supported models.
                - "n": Optional number of outputs to generate for each
input (synthetic data generation)
        output_schema: DEPRECATED. Use 'output' parameter instead.
                       Dictionary defining the expected output structure
for backward compatibility.
        **kwargs: Additional configuration options:
            - model: LLM model to use (default: from config)
            - batch_prompt: Template for processing multiple documents in
a single prompt
            - max_batch_size: Maximum number of documents to process in a
single batch
            - optimize: Flag to enable operation optimization (default:
True)
            - recursively_optimize: Flag to enable recursive optimization
(default: False)
            - sample: Number of samples to use for the operation
            - tools: List of tool definitions for LLM use
            - validate: List of Python expressions to validate output
            - num_retries_on_validate_failure: Number of retry attempts
(default: 0)
            - gleaning: Configuration for LLM-based refinement
            - drop_keys: List of keys to drop from input
            - timeout: Timeout for each LLM call in seconds (default:
120)
            - max_retries_per_timeout: Maximum retries per timeout
(default: 2)
            - litellm_completion_kwargs: Additional parameters for
LiteLLM
            - skip_on_error: Skip operation if LLM returns error
(default: False)
            - bypass_cache: Bypass cache for this operation (default:
```

```
False)
            - n: Number of outputs to generate for each input (synthetic
data generation)

    Returns:
        pd.DataFrame: A new DataFrame containing the transformed data
with columns
                      matching the output schema.

    Examples:
        >>> # Extract entities and sentiment
        >>> df.semantic.map(
        ...     prompt="Analyze this text: {{input.text}}",
        ...     output={
        ...         "schema": {
        ...             "entities": "list[str]",
        ...             "sentiment": "str"
        ...         }
        ...     },
        ...     validate=["len(output['entities']) <= 5"],
        ...     num_retries_on_validate_failure=2
        ... )

        >>> # Generate synthetic data with multiple variations per input
        >>> df.semantic.map(
        ...     prompt="Create a headline for: {{input.topic}}",
        ...     output={"schema": {"headline": "str"}, "n": 5}
        ... )

        >>> # Use structured output mode for better JSON schema support
        >>> df.semantic.map(
        ...     prompt="Extract structured data: {{input.text}}",
        ...     output={
        ...         "schema": {"name": "str", "age": "int", "tags":
"list[str]"},
        ...         "mode": "structured_output"
        ...     }
        ... )
    """
    # Convert DataFrame to list of dicts for DocETL
    input_data = self._df.to_dict("records")

    # Handle backward compatibility: if output_schema is provided,
convert it to output format
    if output_schema is not None and output is None:
        output = {"schema": output_schema}
        if "n" in kwargs:
            output["n"] = kwargs.pop("n")
    elif output is None and output_schema is None:
        raise ValueError("Either 'output' or 'output_schema' must be
provided")
    elif output is not None and output_schema is not None:
        raise ValueError(
            "Cannot provide both 'output' and 'output_schema' parameters"
        )

    # Validate output parameter
    if not isinstance(output, dict):
        raise ValueError("output must be a dictionary")

    if "schema" not in output:
```

```
            raise ValueError("output must contain a 'schema' key")

        # Validate output mode if provided
        output_mode = output.get("mode", "tools")
        if output_mode not in ["tools", "structured_output"]:
            raise ValueError(
                f"output mode must be 'tools' or 'structured_output', got
'{output_mode}'"
            )

        # Create map operation config
        map_config = {
            "type": "map",
            "name": f"semantic_map_{len(self._history)}",
            "prompt": prompt,
            "output": output,
            **kwargs,
        }

        # Create and execute map operation
        map_op = MapOperation(
            runner=self.runner,
            config=map_config,
            default_model=self.runner.config["default_model"],
            max_threads=self.runner.max_threads,
            console=self.runner.console,
            status=self.runner.status,
        )
        results, cost = map_op.execute(input_data)

        return self._record_operation(results, "map", map_config, cost)
```

Example usage:

```
# Basic map operation
df.semantic.map(
    prompt="Extract sentiment and key points from: {{input.text}}",
    output={
        "schema": {
            "sentiment": "str",
            "key_points": "list[str]"
        }
    },
    validate=["len(output['key_points']) <= 5"],
    num_retries_on_validate_failure=2
)

# Using structured output mode for better JSON schema support
df.semantic.map(
    prompt="Extract detailed information from: {{input.text}}",
    output={
        "schema": {
            "company": "str",
            "product": "str",
            "features": "list[str]"
        },
        "mode": "structured_output"
    }
```

```
)

# Backward compatible syntax (still supported)
df.semantic.map(
    prompt="Extract sentiment from: {{input.text}}",
    output_schema={"sentiment": "str"}
)
```

Filter Operation

Filter DataFrame rows based on semantic conditions.

Documentation: https://ucbepic.github.io/docetl/operators/filter/

Parameters:

Name	Type	Description	Default
`prompt`	`str`	Jinja template string for generating prompts	*required*
`output`	`dict[str, Any] | None`	Output configuration with keys: - "schema": Dictionary defining the expected output structure and types For filtering, this must be a single boolean field (e.g., {"keep": "bool"}) - "mode": Optional output mode. Either "tools" (default) or "structured_output"	`None`
`output_schema`	`dict[str, Any] | None`	DEPRECATED. Use 'output' parameter instead. Backward-compatible schema dict.	`None`
`**kwargs`		Additional configuration options: - model: LLM model to use - validate: List of validation expressions - num_retries_on_validate_failure: Number of retries - timeout: Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2) - skip_on_error: Skip rows on LLM error (default: False) -	`{}`

Name	Type	Description	Default
		bypass_cache: Bypass cache for this operation (default: False)	

Returns:

Type	Description
`DataFrame`	pd.DataFrame: Filtered DataFrame containing only rows where the model returned True

Examples:

```
>>> # Simple filtering
>>> df.semantic.filter(
...     prompt="Is this about technology? {{input.text}}"
... )
```

```
>>> # Custom output schema
>>> df.semantic.filter(
...     prompt="Analyze if this is relevant: {{input.text}}",
...     output={"schema": {"keep": "bool", "reason": "str"}}
... )
```

Source code in `docetl/apis/pd_accessors.py`

```python
def filter(
    self,
    prompt: str,
    *,
    output: dict[str, Any] | None = None,
    output_schema: dict[str, Any] | None = None,
    **kwargs,
) -> pd.DataFrame:
    """
    Filter DataFrame rows based on semantic conditions.

    Documentation: https://ucbepic.github.io/docetl/operators/filter/

    Args:
        prompt: Jinja template string for generating prompts
        output: Output configuration with keys:
            - "schema": Dictionary defining the expected output structure
and types
              For filtering, this must be a single boolean field (e.g.,
{"keep": "bool"})
            - "mode": Optional output mode. Either "tools" (default) or
"structured_output"
        output_schema: DEPRECATED. Use 'output' parameter instead.
Backward-compatible schema dict.
        **kwargs: Additional configuration options:
            - model: LLM model to use
            - validate: List of validation expressions
            - num_retries_on_validate_failure: Number of retries
            - timeout: Timeout in seconds (default: 120)
            - max_retries_per_timeout: Max retries per timeout (default:
2)
            - skip_on_error: Skip rows on LLM error (default: False)
            - bypass_cache: Bypass cache for this operation (default:
False)

    Returns:
        pd.DataFrame: Filtered DataFrame containing only rows where the
model
                      returned True

    Examples:
        >>> # Simple filtering
        >>> df.semantic.filter(
        ...     prompt="Is this about technology? {{input.text}}"
        ... )

        >>> # Custom output schema
        >>> df.semantic.filter(
        ...     prompt="Analyze if this is relevant: {{input.text}}",
        ...     output={"schema": {"keep": "bool", "reason": "str"}}
        ... )
    """
    # Convert DataFrame to list of dicts
    input_data = self._df.to_dict("records")

    # Backward compatibility and defaults
    if output is None and output_schema is not None:
```

```
            output = {"schema": output_schema}
        if output is None and output_schema is None:
            output = {"schema": {"keep": "bool"}}

        # Validate output
        if not isinstance(output, dict):
            raise ValueError("output must be a dictionary")
        if "schema" not in output:
            raise ValueError("output must contain a 'schema' key")
        output_mode = output.get("mode", "tools")
        if output_mode not in ["tools", "structured_output"]:
            raise ValueError(
                f"output mode must be 'tools' or 'structured_output', got
'{output_mode}'"
            )

        # Create map operation config for filtering
        filter_config = {
            "type": "map",
            "name": f"semantic_filter_{len(self._history)}",
            "prompt": prompt,
            "output": output,
            **kwargs,
        }

        # Create and execute filter operation
        filter_op = FilterOperation(
            runner=self.runner,
            config=filter_config,
            default_model=self.runner.config["default_model"],
            max_threads=self.runner.max_threads,
            console=self.runner.console,
            status=self.runner.status,
        )
        results, cost = filter_op.execute(input_data)

        return self._record_operation(results, "filter", filter_config, cost)
```

Example usage:

```
# Simple filtering
df.semantic.filter(
    prompt="Is this text about technology? {{input.text}}"
)

# Custom output with reasons
df.semantic.filter(
    prompt="Analyze if this is relevant: {{input.text}}",
    output={
        "schema": {
            "keep": "bool",
            "reason": "str"
        }
    }
)
```

Merge Operation (Experimental)

> **Note**: The merge operation is an experimental feature based on our equijoin operator. It provides a pandas-like interface for semantic record matching and deduplication. When `fuzzy=True`, it automatically invokes optimization to improve performance while maintaining accuracy.

Semantically merge two DataFrames based on flexible matching criteria.

Documentation: https://ucbepic.github.io/docetl/operators/equijoin/

When fuzzy=True and no blocking parameters are provided, this method automatically invokes the JoinOptimizer to generate efficient blocking conditions. The optimizer will suggest blocking thresholds and conditions to improve performance while maintaining the target recall. The optimized configuration will be displayed for future reuse.

Parameters:

Name	Type	Description	Default
`right`	`DataFrame`	Right DataFrame to merge with	*required*
`comparison_prompt`	`str`	Prompt template for comparing records	*required*
`fuzzy`	`bool`	Whether to use fuzzy matching with optimization (default: False)	`False`
`**kwargs`		Additional configuration options: - model: LLM model to use - blocking_threshold: Threshold for blocking optimization - blocking_conditions: Custom blocking conditions - target_recall: Target recall for optimization (default: 0.95) - estimated_selectivity: Estimated match rate - validate: List of validation expressions - num_retries_on_validate_failure: Number of retries - timeout:	`{}`

Name	Type	Description	Default
		Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2)	

Returns:

Type	Description
`DataFrame`	pd.DataFrame: Merged DataFrame containing matched records

Examples:

```
>>> # Simple merge
>>> merged_df = df1.semantic.merge(
...     df2,
...     comparison_prompt="Are these records about the same entity? {{input1}} vs {{input2}}"
... )
```

```
>>> # Fuzzy merge with automatic optimization
>>> merged_df = df1.semantic.merge(
...     df2,
...     comparison_prompt="Compare: {{input1}} vs {{input2}}",
...     fuzzy=True,
...     target_recall=0.9
... )
```

```
>>> # Fuzzy merge with manual blocking parameters
>>> merged_df = df1.semantic.merge(
...     df2,
...     comparison_prompt="Compare: {{input1}} vs {{input2}}",
...     fuzzy=False,
...     blocking_threshold=0.8,
...     blocking_conditions=["input1.category == input2.category"]
... )
```

Source code in `docetl/apis/pd_accessors.py`

```
299  def merge(
300      self,
301      right: pd.DataFrame,
302      comparison_prompt: str,
303      *,
304      fuzzy: bool = False,
305      **kwargs,
306  ) -> pd.DataFrame:
307      """
308      Semantically merge two DataFrames based on flexible matching
309  criteria.
310
311      Documentation: https://ucbepic.github.io/docetl/operators/equijoin/
312
313      When fuzzy=True and no blocking parameters are provided, this method
314  automatically
315      invokes the JoinOptimizer to generate efficient blocking conditions.
316  The optimizer
317      will suggest blocking thresholds and conditions to improve
318  performance while
319      maintaining the target recall. The optimized configuration will be
320  displayed
321      for future reuse.
322
323      Args:
324          right: Right DataFrame to merge with
325          comparison_prompt: Prompt template for comparing records
326          fuzzy: Whether to use fuzzy matching with optimization (default:
327  False)
328          **kwargs: Additional configuration options:
329              - model: LLM model to use
330              - blocking_threshold: Threshold for blocking optimization
331              - blocking_conditions: Custom blocking conditions
332              - target_recall: Target recall for optimization (default:
333  0.95)
334              - estimated_selectivity: Estimated match rate
335              - validate: List of validation expressions
336              - num_retries_on_validate_failure: Number of retries
337              - timeout: Timeout in seconds (default: 120)
338              - max_retries_per_timeout: Max retries per timeout (default:
339  2)
340
341      Returns:
342          pd.DataFrame: Merged DataFrame containing matched records
343
344      Examples:
345          >>> # Simple merge
346          >>> merged_df = df1.semantic.merge(
347          ...     df2,
348          ...     comparison_prompt="Are these records about the same
349  entity? {{input1}} vs {{input2}}"
350          ... )
351
352          >>> # Fuzzy merge with automatic optimization
353          >>> merged_df = df1.semantic.merge(
354          ...     df2,
355          ...     comparison_prompt="Compare: {{input1}} vs {{input2}}",
```

```
        ...     fuzzy=True,
        ...     target_recall=0.9
        ... )

        >>> # Fuzzy merge with manual blocking parameters
        >>> merged_df = df1.semantic.merge(
        ...     df2,
        ...     comparison_prompt="Compare: {{input1}} vs {{input2}}",
        ...     fuzzy=False,
        ...     blocking_threshold=0.8,
        ...     blocking_conditions=["input1.category ==
input2.category"]
        ... )
    """
    # Convert DataFrames to lists of dicts
    left_data = self._df.to_dict("records")
    right_data = right.to_dict("records")

    # Create equijoin operation config
    join_config = {
        "type": "equijoin",
        "name": f"semantic_merge_{len(self._history)}",
        "comparison_prompt": comparison_prompt,
        **kwargs,
    }

    # If fuzzy matching and no blocking params provided, use
JoinOptimizer
    if (
        fuzzy
        and not kwargs.get("blocking_threshold")
        and not kwargs.get("blocking_conditions")
    ):
        join_optimizer = JoinOptimizer(
            self.runner,
            join_config,
            target_recall=kwargs.get("target_recall", 0.95),
            estimated_selectivity=kwargs.get("estimated_selectivity",
None),
        )
        optimized_config, optimizer_cost, _ =
join_optimizer.optimize_equijoin(
            left_data, right_data, skip_map_gen=True,
skip_containment_gen=True
        )

        # Print optimized config for reuse
        self.runner.console.log(
            Panel.fit(
                "[bold cyan]Optimized Configuration for Merge[/bold
cyan]\n"
                "[yellow]Consider adding these parameters to avoid re-
running optimization:[/yellow]\n\n"
                f"blocking_threshold:
{optimized_config.get('blocking_threshold')}\n"
                f"blocking_keys:
{optimized_config.get('blocking_keys')}\n"
                f"blocking_conditions:
{optimized_config.get('blocking_conditions', [])}\n",
                title="Optimization Results",
            )
```

```
            )
            join_config = optimized_config
            optimizer_cost_value = optimizer_cost
        else:
            optimizer_cost_value = 0.0

        # Create and execute equijoin operation
        join_op = EquijoinOperation(
            runner=self.runner,
            config=join_config,
            default_model=self.runner.config["default_model"],
            max_threads=self.runner.max_threads,
            console=self.runner.console,
            status=self.runner.status,
        )
        results, cost = join_op.execute(left_data, right_data)

        return self._record_operation(
            results, "equijoin", join_config, cost + optimizer_cost_value
        )
```

Example usage:

```
# Simple merge
merged_df = df1.semantic.merge(
    df2,
    comparison_prompt="Are these records about the same entity? {{input1}} vs
{{input2}}"
)

# Fuzzy merge with optimization
merged_df = df1.semantic.merge(
    df2,
    comparison_prompt="Compare: {{input1}} vs {{input2}}",
    fuzzy=True,
    target_recall=0.9
)
```

Aggregate Operation

Semantically aggregate data with optional fuzzy matching.

Documentation: - Resolve Operation: https://ucbepic.github.io/docetl/operators/resolve/ - Reduce Operation: https://ucbepic.github.io/docetl/operators/reduce/

When fuzzy=True and no blocking parameters are provided in resolve_kwargs, this method automatically invokes the JoinOptimizer to generate efficient blocking conditions for the resolve phase. The optimizer will suggest blocking thresholds and conditions to improve performance while maintaining the target recall. The optimized configuration will be displayed for future reuse.

The resolve phase is skipped if: - fuzzy=False - reduce_keys=["_all"] - input data has 5 or fewer rows

Parameters:

Name	Type	Description	Default
`reduce_prompt`	`str`	Prompt template for the reduction phase	*required*
`output`	`dict[str, Any] | None`	Output configuration with keys: - "schema": Dictionary defining the expected output structure and types - "mode": Optional output mode. Either "tools" (default) or "structured_output"	`None`
`output_schema`	`dict[str, Any] | None`	DEPRECATED. Use 'output' parameter instead. Backward-compatible schema dict	`None`
`fuzzy`	`bool`	Whether to use fuzzy matching for resolution (default: False)	`False`
`comparison_prompt`	`str | None`	Prompt template for comparing records during resolution	`None`
`resolution_prompt`	`str | None`	Prompt template for resolving conflicts	`None`
`resolution_output`	`dict[str, Any] | None`	Output configuration for resolution (new API with schema key)	`None`
`resolution_output_schema`	`dict[str, Any] | None`	Schema for resolution output (deprecated, use resolution_output)	`None`
`reduce_keys`	`str | list[str]`	Keys to group by for reduction (default: ["_all"])	`['_all']`

Name	Type	Description	Default
`resolve_kwargs`	`dict[str, Any]`	Additional kwargs for resolve operation: - model: LLM model to use - blocking_threshold: Threshold for blocking optimization - blocking_conditions: Custom blocking conditions - target_recall: Target recall for optimization (default: 0.95) - estimated_selectivity: Estimated match rate - validate: List of validation expressions - num_retries_on_validate_failure: Number of retries - timeout: Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2)	`{}`
`reduce_kwargs`	`dict[str, Any]`	Additional kwargs for reduce operation: - model: LLM model to use - validate: List of validation expressions - num_retries_on_validate_failure: Number of retries - timeout: Timeout in seconds (default: 120) - max_retries_per_timeout: Max retries per timeout (default: 2)	`{}`

Returns:

Type	Description
`DataFrame`	pd.DataFrame: Aggregated DataFrame with columns matching output['schema']

Examples:

```
>>> # Simple aggregation
>>> df.semantic.agg(
...     reduce_prompt="Summarize these items: {{input.text}}",
...     output={"schema": {"summary": "str"}}
... )
```

```
>>> # Fuzzy matching with automatic optimization
>>> df.semantic.agg(
...     reduce_prompt="Combine these items: {{input.text}}",
...     output={"schema": {"combined": "str"}},
...     fuzzy=True,
...     comparison_prompt="Are these items similar: {{input1.text}} vs
{{input2.text}}",
...     resolution_prompt="Resolve conflicts between: {{items}}",
...     resolution_output={"schema": {"resolved": "str"}}
... )
```

```
>>> # Fuzzy matching with manual blocking parameters
>>> df.semantic.agg(
...     reduce_prompt="Combine these items: {{input.text}}",
...     output={"schema": {"combined": "str"}},
...     fuzzy=False,
...     comparison_prompt="Compare items: {{input1.text}} vs {{input2.text}}",
...     resolve_kwargs={
...         "blocking_threshold": 0.8,
...         "blocking_conditions": ["input1.category == input2.category"]
...     }
... )
```

Source code in `docetl/apis/pd_accessors.py`

```
    def agg(
        self,
        *,
        # Reduction phase params (required)
        reduce_prompt: str,
        output: dict[str, Any] | None = None,
        output_schema: dict[str, Any] | None = None,
        # Resolution and reduce phase params (optional)
        fuzzy: bool = False,
        comparison_prompt: str | None = None,
        resolution_prompt: str | None = None,
        resolution_output: dict[str, Any] | None = None,
        resolution_output_schema: dict[str, Any] | None = None,
        reduce_keys: str | list[str] = ["_all"],
        resolve_kwargs: dict[str, Any] = {},
        reduce_kwargs: dict[str, Any] = {},
    ) -> pd.DataFrame:
        """
        Semantically aggregate data with optional fuzzy matching.

        Documentation:
        - Resolve Operation:
https://ucbepic.github.io/docetl/operators/resolve/
        - Reduce Operation:
https://ucbepic.github.io/docetl/operators/reduce/

        When fuzzy=True and no blocking parameters are provided in
resolve_kwargs,
        this method automatically invokes the JoinOptimizer to generate
efficient
        blocking conditions for the resolve phase. The optimizer will
suggest
        blocking thresholds and conditions to improve performance while
maintaining
        the target recall. The optimized configuration will be displayed
for future reuse.

        The resolve phase is skipped if:
        - fuzzy=False
        - reduce_keys=["_all"]
        - input data has 5 or fewer rows

        Args:
            reduce_prompt: Prompt template for the reduction phase
            output: Output configuration with keys:
                - "schema": Dictionary defining the expected output
structure and types
                - "mode": Optional output mode. Either "tools" (default)
or "structured_output"
            output_schema: DEPRECATED. Use 'output' parameter instead.
Backward-compatible schema dict
            fuzzy: Whether to use fuzzy matching for resolution (default:
False)
            comparison_prompt: Prompt template for comparing records
during resolution
            resolution_prompt: Prompt template for resolving conflicts
            resolution_output: Output configuration for resolution (new
```

```
API with schema key)
            resolution_output_schema: Schema for resolution output
(deprecated, use resolution_output)
            reduce_keys: Keys to group by for reduction (default:
["_all"])
            resolve_kwargs: Additional kwargs for resolve operation:
                - model: LLM model to use
                - blocking_threshold: Threshold for blocking optimization
                - blocking_conditions: Custom blocking conditions
                - target_recall: Target recall for optimization (default:
0.95)
                - estimated_selectivity: Estimated match rate
                - validate: List of validation expressions
                - num_retries_on_validate_failure: Number of retries
                - timeout: Timeout in seconds (default: 120)
                - max_retries_per_timeout: Max retries per timeout
(default: 2)
            reduce_kwargs: Additional kwargs for reduce operation:
                - model: LLM model to use
                - validate: List of validation expressions
                - num_retries_on_validate_failure: Number of retries
                - timeout: Timeout in seconds (default: 120)
                - max_retries_per_timeout: Max retries per timeout
(default: 2)

        Returns:
            pd.DataFrame: Aggregated DataFrame with columns matching
output['schema']

        Examples:
            >>> # Simple aggregation
            >>> df.semantic.agg(
            ...     reduce_prompt="Summarize these items:
{{input.text}}",
            ...     output={"schema": {"summary": "str"}}
            ... )

            >>> # Fuzzy matching with automatic optimization
            >>> df.semantic.agg(
            ...     reduce_prompt="Combine these items: {{input.text}}",
            ...     output={"schema": {"combined": "str"}},
            ...     fuzzy=True,
            ...     comparison_prompt="Are these items similar:
{{input1.text}} vs {{input2.text}}",
            ...     resolution_prompt="Resolve conflicts between:
{{items}}",
            ...     resolution_output={"schema": {"resolved": "str"}}
            ... )

            >>> # Fuzzy matching with manual blocking parameters
            >>> df.semantic.agg(
            ...     reduce_prompt="Combine these items: {{input.text}}",
            ...     output={"schema": {"combined": "str"}},
            ...     fuzzy=False,
            ...     comparison_prompt="Compare items: {{input1.text}} vs
{{input2.text}}",
            ...     resolve_kwargs={
            ...         "blocking_threshold": 0.8,
            ...         "blocking_conditions": ["input1.category ==
input2.category"]
            ...     }
```

```
            ... )
        """
        # Convert DataFrame to list of dicts
        input_data = self._df.to_dict("records")

        # Handle backward compatibility: if output_schema is provided,
convert it to output format
        if output_schema is not None and output is None:
            output = {"schema": output_schema}
        elif output is None and output_schema is None:
            raise ValueError("Either 'output' or 'output_schema' must be
provided")
        elif output is not None and output_schema is not None:
            raise ValueError(
                "Cannot provide both 'output' and 'output_schema'
parameters"
            )

        # Validate output parameter
        if not isinstance(output, dict):
            raise ValueError("output must be a dictionary")
        if "schema" not in output:
            raise ValueError("output must contain a 'schema' key")

        # Handle backward compatibility for resolution_output_schema
        if resolution_output_schema is not None and resolution_output is
None:
            resolution_output = {"schema": resolution_output_schema}
        elif resolution_output is not None and resolution_output_schema
is not None:
            raise ValueError(
                "Cannot provide both 'resolution_output' and
'resolution_output_schema' parameters"
            )

        # Validate output mode if provided
        output_mode = output.get("mode", "tools")
        if output_mode not in ["tools", "structured_output"]:
            raise ValueError(
                f"output mode must be 'tools' or 'structured_output', got
'{output_mode}'"
            )

        # Change keys to list
        if isinstance(reduce_keys, str):
            reduce_keys = [reduce_keys]

        # Skip resolution if using _all or fuzzy is False
        if reduce_keys == ["_all"] or not fuzzy or len(input_data) <= 5:
            resolved_data = input_data
        else:
            # Synthesize comparison prompt if not provided
            if comparison_prompt is None:
                # Build record template from reduce_keys
                record_template = ", ".join(
                    f"{key}: {{{{ input{0}.{key} }}}}" for key in
reduce_keys
                )

                # Add context about how fields were created
                context =
```

```
self._synthesize_comparison_context(reduce_keys)

                comparison_prompt = f"""Do the following two records
represent the same concept? Your answer should be true or false.{context}

Record 1: {record_template.replace('input0', 'input1')}
Record 2: {record_template.replace('input0', 'input2')}"""

            # Configure resolution
            resolve_config = {
                "type": "resolve",
                "name": f"semantic_resolve_{len(self._history)}",
                "comparison_prompt": comparison_prompt,
                **resolve_kwargs,
            }

            # Add resolution prompt and schema if provided
            if resolution_prompt:
                resolve_config["resolution_prompt"] = resolution_prompt
                if resolution_output:
                    # Use the new resolution_output format
                    resolve_config["output"] = resolution_output
                    if "keys" not in resolve_config["output"]:
                        # Add keys from schema if not explicitly provided
                        resolve_config["output"]["keys"] = list(
                            resolution_output["schema"].keys()
                        )
                else:
                    # No resolution output provided, use reduce_keys
                    resolve_config["output"] = {"keys": reduce_keys}
            else:
                resolve_config["output"] = {"keys": reduce_keys}

            # If blocking params not provided, use JoinOptimizer to
synthesize them
            if not resolve_kwargs or (
                "blocking_threshold" not in resolve_kwargs
                and "blocking_conditions" not in resolve_kwargs
            ):
                join_optimizer = JoinOptimizer(
                    self.runner,
                    resolve_config,
                    target_recall=(
                        resolve_kwargs.get("target_recall", 0.95)
                        if resolve_kwargs
                        else 0.95
                    ),
                    estimated_selectivity=(
                        resolve_kwargs.get("estimated_selectivity", None)
                        if resolve_kwargs
                        else None
                    ),
                )
                optimized_config, optimizer_cost =
join_optimizer.optimize_resolve(
                    input_data
                )

                # Print optimized config for reuse
                self.runner.console.log(
                    Panel.fit(
```

```
                        "[bold cyan]Optimized Configuration for
Resolve[/bold cyan]\n"
                        "[yellow]Consider adding these parameters to
avoid re-running optimization:[/yellow]\n\n"
                        f"blocking_threshold:
{optimized_config.get('blocking_threshold')}\n"
                        f"blocking_keys:
{optimized_config.get('blocking_keys')}\n"
                        f"blocking_conditions:
{optimized_config.get('blocking_conditions', [])}\n",
                        title="Optimization Results",
                    )
                )
            else:
                # Use provided blocking params
                optimized_config = resolve_config.copy()
                optimizer_cost = 0.0

                # Execute resolution with optimized config
                resolve_op = ResolveOperation(
                    runner=self.runner,
                    config=optimized_config,
                    default_model=self.runner.config["default_model"],
                    max_threads=self.runner.max_threads,
                    console=self.runner.console,
                    status=self.runner.status,
                )
                resolved_data, resolve_cost = resolve_op.execute(input_data)
                _ = self._record_operation(
                    resolved_data,
                    "resolve",
                    optimized_config,
                    resolve_cost + optimizer_cost,
                )

        # Configure reduction
        reduce_config = {
            "type": "reduce",
            "name": f"semantic_reduce_{len(self._history)}",
            "reduce_key": reduce_keys,
            "prompt": reduce_prompt,
            "output": output,
            **reduce_kwargs,
        }

        # Execute reduction
        reduce_op = ReduceOperation(
            runner=self.runner,
            config=reduce_config,
            default_model=self.runner.config["default_model"],
            max_threads=self.runner.max_threads,
            console=self.runner.console,
            status=self.runner.status,
        )
        results, reduce_cost = reduce_op.execute(resolved_data)

        return self._record_operation(results, "reduce", reduce_config,
    reduce_cost)
```

Example usage:

```
# Simple aggregation
df.semantic.agg(
    reduce_prompt="Summarize these items: {{input.text}}",
    output={"schema": {"summary": "str"}}
)

# Fuzzy matching with custom resolution
df.semantic.agg(
    reduce_prompt="Combine these items: {{input.text}}",
    output={"schema": {"combined": "str"}},
    fuzzy=True,
    comparison_prompt="Are these items similar: {{input1.text}} vs
{{input2.text}}",
    resolution_prompt="Resolve conflicts between: {{items}}",
    resolution_output={"schema": {"resolved": "str"}}
)
```

Split Operation

```
    Split DataFrame rows into multiple chunks based on content.

    Documentation: https://ucbepic.github.io/docetl/operators/split/

    Args:
        split_key: The column containing content to split
        method: Splitting method, either "token_count" or "delimiter"
        method_kwargs: Dictionary containing method-specific parameters:
            - For "token_count": {"num_tokens": int, "model": str (optional)}
            - For "delimiter": {"delimiter": str, "num_splits_to_group": int
(optional)}
        **kwargs: Additional configuration options:
            - model: LLM model to use for tokenization (default: from config)

    Returns:
        pd.DataFrame: DataFrame with split content, including:
            - {split_key}_chunk: The content of each chunk
            - {operation_name}_id: Unique identifier for the original document
            - {operation_name}_chunk_num: Sequential chunk number within the
document

    Examples:
        >>> # Split by token count
        >>> df.semantic.split(
        ...     split_key="content",
        ...     method="token_count",
        ...     method_kwargs={"num_tokens": 100}
        ... )

        >>> # Split by delimiter
        >>> df.semantic.split(
        ...     split_key="text",
        ...     method="delimiter",
        ...     method_kwargs={"delimiter": "
```

", "num_splits_to_group": 2} ...)

Source code in `docetl/apis/pd_accessors.py`

```python
def split(
    self, split_key: str, method: str, method_kwargs: dict[str, Any],
**kwargs
) -> pd.DataFrame:
    """
    Split DataFrame rows into multiple chunks based on content.

    Documentation: https://ucbepic.github.io/docetl/operators/split/

    Args:
        split_key: The column containing content to split
        method: Splitting method, either "token_count" or "delimiter"
        method_kwargs: Dictionary containing method-specific parameters:
            - For "token_count": {"num_tokens": int, "model": str
(optional)}
            - For "delimiter": {"delimiter": str, "num_splits_to_group":
int (optional)}
        **kwargs: Additional configuration options:
            - model: LLM model to use for tokenization (default: from
config)

    Returns:
        pd.DataFrame: DataFrame with split content, including:
            - {split_key}_chunk: The content of each chunk
            - {operation_name}_id: Unique identifier for the original
document
            - {operation_name}_chunk_num: Sequential chunk number within
the document

    Examples:
        >>> # Split by token count
        >>> df.semantic.split(
        ...     split_key="content",
        ...     method="token_count",
        ...     method_kwargs={"num_tokens": 100}
        ... )

        >>> # Split by delimiter
        >>> df.semantic.split(
        ...     split_key="text",
        ...     method="delimiter",
        ...     method_kwargs={"delimiter": "\n\n",
"num_splits_to_group": 2}
        ... )
    """
    # Convert DataFrame to list of dicts
    input_data = self._df.to_dict("records")

    # Create split operation config
    split_config = {
        "type": "split",
        "name": f"semantic_split_{len(self._history)}",
        "split_key": split_key,
        "method": method,
        "method_kwargs": method_kwargs,
        **kwargs,
    }
```

```
820
821         # Create and execute split operation
822         split_op = SplitOperation(
823             runner=self.runner,
824             config=split_config,
825             default_model=self.runner.config["default_model"],
                max_threads=self.runner.max_threads,
                console=self.runner.console,
                status=self.runner.status,
            )
            results, cost = split_op.execute(input_data)

            return self._record_operation(results, "split", split_config, cost)
```

Example usage:

```
# Split by token count
df.semantic.split(
    split_key="content",
    method="token_count",
    method_kwargs={"num_tokens": 100}
)

# Split by delimiter
df.semantic.split(
    split_key="text",
    method="delimiter",
    method_kwargs={"delimiter": "\n\n", "num_splits_to_group": 2}
)
```

Gather Operation

Gather contextual information from surrounding chunks to enhance each chunk.

Documentation: https://ucbepic.github.io/docetl/operators/gather/

Parameters:

Name	Type	Description	Default
`content_key`	`str`	The column containing the main content to be enhanced	*required*
`doc_id_key`	`str`	The column containing document identifiers to group chunks	*required*
`order_key`	`str`	The column containing chunk order numbers within	*required*

Name	Type	Description	Default
		documents	
`peripheral_chunks`	`dict[str, Any] | None`	Configuration for surrounding context: - previous: {"head": {"count": int}, "tail": {"count": int}, "middle": {}} - next: {"head": {"count": int}, "tail": {"count": int}, "middle": {}}	`None`
`**kwargs`		Additional configuration options: - main_chunk_start: Start marker for main chunk (default: "--- Begin Main Chunk ---") - main_chunk_end: End marker for main chunk (default: "--- End Main Chunk ---") - doc_header_key: Column containing document headers (optional)	`{}`

Returns:

Type	Description
`DataFrame`	pd.DataFrame: DataFrame with enhanced content including: - {content_key}_rendered: The main content with surrounding context

Examples:

```
>>> # Basic gathering with surrounding context
>>> df.semantic.gather(
...     content_key="chunk_content",
...     doc_id_key="document_id",
...     order_key="chunk_number",
...     peripheral_chunks={
...         "previous": {"head": {"count": 2}, "tail": {"count": 1}},
...         "next": {"head": {"count": 1}, "tail": {"count": 2}}
...     }
... )
```

```
>>> # Simple gathering without peripheral chunks
>>> df.semantic.gather(
...     content_key="content",
...     doc_id_key="doc_id",
...     order_key="order"
... )
```

Source code in `docetl/apis/pd_accessors.py`

```
827  def gather(
828      self,
829      content_key: str,
830      doc_id_key: str,
831      order_key: str,
832      peripheral_chunks: dict[str, Any] | None = None,
833      **kwargs,
834  ) -> pd.DataFrame:
835      """
836      Gather contextual information from surrounding chunks to enhance each
837  chunk.
838
839      Documentation: https://ucbepic.github.io/docetl/operators/gather/
840
841      Args:
842          content_key: The column containing the main content to be
843  enhanced
844          doc_id_key: The column containing document identifiers to group
845  chunks
846          order_key: The column containing chunk order numbers within
847  documents
848          peripheral_chunks: Configuration for surrounding context:
849              - previous: {"head": {"count": int}, "tail": {"count": int},
850  "middle": {}}
851              - next: {"head": {"count": int}, "tail": {"count": int},
852  "middle": {}}
853          **kwargs: Additional configuration options:
854              - main_chunk_start: Start marker for main chunk (default: "--
855  - Begin Main Chunk ---")
856              - main_chunk_end: End marker for main chunk (default: "---
857  End Main Chunk ---")
858              - doc_header_key: Column containing document headers
859  (optional)
860
861      Returns:
862          pd.DataFrame: DataFrame with enhanced content including:
863              - {content_key}_rendered: The main content with surrounding
864  context
865
866      Examples:
867          >>> # Basic gathering with surrounding context
868          >>> df.semantic.gather(
869          ...     content_key="chunk_content",
870          ...     doc_id_key="document_id",
871          ...     order_key="chunk_number",
872          ...     peripheral_chunks={
873          ...         "previous": {"head": {"count": 2}, "tail": {"count":
874  1}},
875          ...         "next": {"head": {"count": 1}, "tail": {"count": 2}}
876          ...     }
877          ... )
878
879          >>> # Simple gathering without peripheral chunks
880          >>> df.semantic.gather(
881          ...     content_key="content",
882          ...     doc_id_key="doc_id",
883          ...     order_key="order"
```

```
            ... )
        """
        # Convert DataFrame to list of dicts
        input_data = self._df.to_dict("records")

        # Create gather operation config
        gather_config = {
            "type": "gather",
            "name": f"semantic_gather_{len(self._history)}",
            "content_key": content_key,
            "doc_id_key": doc_id_key,
            "order_key": order_key,
            **kwargs,
        }

        # Add peripheral_chunks config if provided
        if peripheral_chunks is not None:
            gather_config["peripheral_chunks"] = peripheral_chunks

        # Create and execute gather operation
        gather_op = GatherOperation(
            runner=self.runner,
            config=gather_config,
            default_model=self.runner.config["default_model"],
            max_threads=self.runner.max_threads,
            console=self.runner.console,
            status=self.runner.status,
        )
        results, cost = gather_op.execute(input_data)

        return self._record_operation(results, "gather", gather_config, cost)
```

Example usage:

```
# Basic gathering with surrounding context
df.semantic.gather(
    content_key="chunk_content",
    doc_id_key="document_id",
    order_key="chunk_number",
    peripheral_chunks={
        "previous": {"head": {"count": 2}, "tail": {"count": 1}},
        "next": {"head": {"count": 1}, "tail": {"count": 2}}
    }
)

# Simple gathering without peripheral chunks
df.semantic.gather(
    content_key="content",
    doc_id_key="doc_id",
    order_key="order"
)
```

Unnest Operation

Unnest list-like or dictionary values into multiple rows.

Documentation: https://ucbepic.github.io/docetl/operators/unnest/

Parameters:

Name	Type	Description	Default
`unnest_key`	`str`	The column containing list-like or dictionary values to unnest	*required*
`keep_empty`	`bool`	Whether to keep rows with empty/null values (default: False)	`False`
`expand_fields`	`list[str] | None`	For dictionary values, which fields to expand (default: all)	`None`
`recursive`	`bool`	Whether to recursively unnest nested structures (default: False)	`False`
`depth`	`int | None`	Maximum depth for recursive unnesting (default: 1, or unlimited if recursive=True)	`None`
`**kwargs`		Additional configuration options	`{}`

Returns:

Type	Description
`DataFrame`	pd.DataFrame: DataFrame with unnested values, where: - For lists: Each list element becomes a separate row - For dicts: Specified fields are expanded into the parent row

Examples:

```
>>> # Unnest a list column
>>> df.semantic.unnest(
...     unnest_key="tags"
... )
# Input:  [{"id": 1, "tags": ["a", "b"]}]
# Output: [{"id": 1, "tags": "a"}, {"id": 1, "tags": "b"}]
```

```
>>> # Unnest a dictionary column with specific fields
>>> df.semantic.unnest(
```

```
...     unnest_key="user_info",
...     expand_fields=["name", "age"]
... )
# Input:  [{"id": 1, "user_info": {"name": "Alice", "age": 30, "email":
"alice@example.com"}}]
# Output: [{"id": 1, "user_info": {...}, "name": "Alice", "age": 30}]
```

```
>>> # Recursive unnesting
>>> df.semantic.unnest(
...     unnest_key="nested_lists",
...     recursive=True,
...     depth=2
... )
```

Source code in `docetl/apis/pd_accessors.py`

```python
def unnest(
    self,
    unnest_key: str,
    keep_empty: bool = False,
    expand_fields: list[str] | None = None,
    recursive: bool = False,
    depth: int | None = None,
    **kwargs,
) -> pd.DataFrame:
    """
    Unnest list-like or dictionary values into multiple rows.

    Documentation: https://ucbepic.github.io/docetl/operators/unnest/

    Args:
        unnest_key: The column containing list-like or dictionary values
to unnest
        keep_empty: Whether to keep rows with empty/null values (default:
False)
        expand_fields: For dictionary values, which fields to expand
(default: all)
        recursive: Whether to recursively unnest nested structures
(default: False)
        depth: Maximum depth for recursive unnesting (default: 1, or
unlimited if recursive=True)
        **kwargs: Additional configuration options

    Returns:
        pd.DataFrame: DataFrame with unnested values, where:
            - For lists: Each list element becomes a separate row
            - For dicts: Specified fields are expanded into the parent
row

    Examples:
        >>> # Unnest a list column
        >>> df.semantic.unnest(
        ...     unnest_key="tags"
        ... )
        # Input:  [{"id": 1, "tags": ["a", "b"]}]
        # Output: [{"id": 1, "tags": "a"}, {"id": 1, "tags": "b"}]

        >>> # Unnest a dictionary column with specific fields
        >>> df.semantic.unnest(
        ...     unnest_key="user_info",
        ...     expand_fields=["name", "age"]
        ... )
        # Input:  [{"id": 1, "user_info": {"name": "Alice", "age": 30,
"email": "alice@example.com"}}]
        # Output: [{"id": 1, "user_info": {...}, "name": "Alice", "age":
30}]

        >>> # Recursive unnesting
        >>> df.semantic.unnest(
        ...     unnest_key="nested_lists",
        ...     recursive=True,
        ...     depth=2
        ... )
```

```
    """
    # Convert DataFrame to list of dicts
    input_data = self._df.to_dict("records")

    # Create unnest operation config
    unnest_config = {
        "type": "unnest",
        "name": f"semantic_unnest_{len(self._history)}",
        "unnest_key": unnest_key,
        "keep_empty": keep_empty,
        "recursive": recursive,
        **kwargs,
    }

    # Add optional parameters if provided
    if expand_fields is not None:
        unnest_config["expand_fields"] = expand_fields
    if depth is not None:
        unnest_config["depth"] = depth

    # Create and execute unnest operation
    unnest_op = UnnestOperation(
        runner=self.runner,
        config=unnest_config,
        default_model=self.runner.config["default_model"],
        max_threads=self.runner.max_threads,
        console=self.runner.console,
        status=self.runner.status,
    )
    results, cost = unnest_op.execute(input_data)

    return self._record_operation(results, "unnest", unnest_config, cost)
```

Example usage:

```
# Unnest a list column
df.semantic.unnest(unnest_key="tags")

# Unnest a dictionary column with specific fields
df.semantic.unnest(
    unnest_key="user_info",
    expand_fields=["name", "age"]
)

# Recursive unnesting with depth control
df.semantic.unnest(
    unnest_key="nested_lists",
    recursive=True,
    depth=2
)
```

Common Features

All operations support:

1. **Cost Tracking**

```
# After any operation
print(f"Operation cost: ${df.semantic.total_cost}")
```

2. **Operation History**

```
# View operation history
for op in df.semantic.history:
    print(f"{op.op_type}: {op.output_columns}")
```

3. **Validation Rules**

```
# Add validation rules to any  map or filter operation
validate=["len(output['tags']) <= 5", "output['score'] >= 0"]
```

For more details on configuration options and best practices, refer to: - DocETL Best Practices - Pipeline Configuration - Output Schemas

Examples

Here are some demonstrating how to use DocETL's pandas integration for various tasks.

Note that caching is enabled, but intermediate outputs are not persisted like in DocETL's YAML interface.

Example 1: Analyzing Customer Reviews

Extract structured insights from customer reviews:

```
import pandas as pd
from docetl import SemanticAccessor

# Load customer reviews
df = pd.DataFrame({
    "review": [
        "Great laptop, fast processor but battery life could be better",
        "The camera quality is amazing, especially in low light",
        "Keyboard feels cheap and the screen is too dim"
    ]
})

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

# Extract structured insights
result = df.semantic.map(
    prompt="""Analyze this product review and extract:
    1. Mentioned features
    2. Sentiment per feature
    3. Overall sentiment

    Review: {{input.review}}""",
    output={
        "schema": {
            "features": "list[str]",
            "feature_sentiments": "dict[str, str]",
            "overall_sentiment": "str"
        }
    }
)

# Filter for negative reviews
negative_reviews = result.semantic.filter(
    prompt="Is this review predominantly negative? Consider the overall
sentiment and feature sentiments.\n{{input}}"
)
```

Example 2: Deduplicating Customer Records

Identify and merge duplicate customer records using fuzzy matching:

```
# Customer records from two sources
df1 = pd.DataFrame({
    "name": ["John Smith", "Mary Johnson"],
    "email": ["john@email.com", "mary.j@email.com"],
    "address": ["123 Main St", "456 Oak Ave"]
})

df2 = pd.DataFrame({
    "name": ["John A Smith", "Mary Johnson"],
    "email": ["john@email.com", "mary.johnson@email.com"],
    "address": ["123 Main Street", "456 Oak Avenue"]
})

# Merge records with fuzzy matching
merged = df1.semantic.merge(
    df2,
    comparison_prompt="""Compare these customer records and determine if they
represent the same person.
    Consider name variations, email patterns, and address formatting.

    Record 1:
    Name: {{input1.name}}
    Email: {{input1.email}}
    Address: {{input1.address}}

    Record 2:
    Name: {{input2.name}}
    Email: {{input2.email}}
    Address: {{input2.address}}""",
    fuzzy=True, # This will automatically invoke optimization
)
```

Example 3: Topic Analysis of News Articles

Group and summarize news articles by topic:

```
# News articles
df = pd.DataFrame({
    "title": ["Apple's New iPhone Launch", "Tech Giants Face Regulation", "AI
Advances in Healthcare"],
    "content": ["Apple announced...", "Lawmakers propose...", "Researchers
develop..."]
})

# First, use a semantic map to extract the topic from each article
df = df.semantic.map(
    prompt="Extract the topic from this article: {{input.content}}",
    output={"schema": {"topic": "str"}}
)
```

```
# Group similar articles and generate summaries
summaries = df.semantic.agg(
    # First, group similar articles
    fuzzy=True,
    reduce_keys=["topic"],
    comparison_prompt="""Are these articles about the same topic or closely
related topics?

    Article 1:
    Title: {{input1.title}}
    Content: {{input1.content}}

    Article 2:
    Title: {{input2.title}}
    Content: {{input2.content}}""",

    # Then, generate a summary for each group
    reduce_prompt="""Summarize these related articles into a comprehensive
overview:

    Articles:
    {{inputs}}""",

    output={
        "schema": {
            "summary": "str",
            "key_points": "list[str]"
        }
    }
)

# Summaries will be a df with the following columns:
# - topic: str (because this was the reduce_keys)
# - summary: str
# - key_points: list[str]
```

Example 4: Multi-step Analysis Pipeline

Combine multiple operations for complex analysis:

```
# Social media posts
posts = pd.DataFrame({
    "text": ["Just tried the new iPhone 15!", "Having issues with iOS 17",
"Android is better"],
    "timestamp": ["2024-01-01", "2024-01-02", "2024-01-03"]
})

# 1. Extract structured information
analyzed = posts.semantic.map(
    prompt="""Analyze this social media post and extract:
    1. Product mentioned
    2. Sentiment
    3. Issues/Praise points
```

```
    Post: {{input.text}}""",
    output={
        "schema": {
            "product": "str",
            "sentiment": "str",
            "points": "list[str]"
        }
    }
)

# 2. Filter relevant posts
relevant = analyzed.semantic.filter(
    prompt="Is this post about Apple products? {{input}}"
)


# 3. Group by issue and summarize
summaries = relevant.semantic.agg(
    fuzzy=True,
    reduce_keys=["product"],
    comparison_prompt="Do these posts discuss the same product?",
    reduce_prompt="Summarize the feedback about this product",
    output={
        "schema": {
            "summary": "str",
            "frequency": "int",
            "severity": "str"
        }
    }
)

# Summaries will be a df with the following columns:
# - product: str (because this was the reduce_keys)
# - summary: str
# - frequency: int
# - severity: str

# Track total cost
print(f"Total analysis cost: ${summaries.semantic.total_cost}")
```

Example 5: Error Handling and Validation

Implement robust error handling and validation:

```
# Product descriptions
df = pd.DataFrame({
    "description": ["High-performance laptop...", "Wireless earbuds...",
"Invalid/"]
})

try:
    result = df.semantic.map(
        prompt="Extract product specifications from: {{input.description}}.
```

```
There should be at least one feature.",
        output={
            "schema": {
                "category": "str",
                "features": "list[str]",
                "price_range": "enum[budget, mid-range, premium, luxury]"
            }
        },
        # Validation rules
        validate=[
            "len(output['features']) >= 1",
        ],
        # Retry configuration
        num_retries_on_validate_failure=2,
    )

    # Check operation history
    for op in result.semantic.history:
        print(f"Operation: {op.op_type}")
        print(f"Modified columns: {op.output_columns}")

except Exception as e:
    print(f"Error during processing: {e}")
```

Example 6: PDF Analysis

DocETL supports native PDF handling with Claude and Gemini, in map and filter operations. Suppose you have a column in your pandas dataframe with PDF paths (1 path per row), and you want the LLM to do some analysis for each PDF. You can do this by setting the `pdf_url_key` parameter in the map or filter operation.

```
df = pd.DataFrame({
    "PdfPath": ["https://docetlcloudbank.blob.core.windows.net/ntsb-
reports/Report_N617GC.pdf",
"https://docetlcloudbank.blob.core.windows.net/ntsb-
reports/Report_CEN25LA075.pdf"]
})

result_df = df.semantic.map(
    prompt="Summarize the air crash report and determine any contributing
factors",
    output={
        "schema": {"summary": "str", "contributing_factors": "list[str]"}
    },
    pdf_url_key="PdfPath", # This is the column with the PDF paths
)

print(result_df.head()) # The result will have the same number of rows as the
input dataframe, with the summary and contributing factors added
```

Example 7: Synthetic Data Generation

DocETL supports generating multiple outputs for each input using the `n` parameter in the map operation. This is useful for synthetic data generation, A/B testing content variations, or creating multiple alternatives for each input.

```
# Starter concepts for content generation
df = pd.DataFrame({
    "product": ["Smart Watch", "Wireless Earbuds", "Home Security Camera"],
    "target_audience": ["Fitness Enthusiasts", "Music Lovers", "Homeowners"]
})

# Generate 5 marketing headlines for each product-audience combination
variations = df.semantic.map(
    prompt="""Generate a compelling marketing headline for the following
product and target audience:

    Product: {{input.product}}
    Target Audience: {{input.target_audience}}

    The headline should:
    - Be attention-grabbing and memorable
    - Speak directly to the target audience's needs or desires
    - Highlight the key benefit of the product
    - Be between 5-10 words
    """,
    output={"schema": {"headline": "str"}, "n": 5}  # Generate 5 variations
for each input row
)

print(f"Original dataframe rows: {len(df)}")
print(f"Generated variations: {len(variations)}")  # Should be 5x the original
count

# Variations dataframe will contain:
# - All original columns (product, target_audience)
# - The new headline column
# - 5 rows for each original row (15 total for this example)

# You can also combine this with other operations
# Filter to only keep the best headlines
best_headlines = variations.semantic.filter(
    prompt="""Is this headline exceptional and likely to drive high
engagement?

    Product: {{input.product}}
    Target Audience: {{input.target_audience}}
    Headline: {{input.headline}}

    Consider catchiness, emotional appeal, and clarity.
    """
)
```

This example will generate 15 total variations (3 products × 5 variations each). You can adjust the `n` parameter to generate more or fewer variations as needed.

Example 8: Document Processing with Split and Gather

Process long documents by splitting them into chunks and adding contextual information:

```
# Long documents that need to be processed in chunks
df = pd.DataFrame({
    "document_id": ["doc1", "doc2"],
    "title": ["Technical Manual", "Research Paper"],
    "content": [
        "Chapter 1: Introduction\n\nThis manual provides comprehensive
guidance for system installation and configuration. The installation process
involves several critical steps that must be followed in order.\n\nChapter 2:
Prerequisites\n\nBefore beginning installation, ensure all system requirements
are met. This includes hardware specifications, software dependencies, and
network connectivity.\n\nChapter 3: Installation\n\nThe installation wizard
guides users through the setup process. Select appropriate configuration
options based on your deployment environment.",
        "Abstract\n\nThis research investigates the impact of machine learning
on data processing efficiency. Our methodology involved testing multiple
algorithms across diverse datasets.\n\nIntroduction\n\nMachine learning has
revolutionized data processing across industries. Previous studies have shown
significant improvements in processing speed and
accuracy.\n\nMethodology\n\nWe conducted experiments using three different ML
algorithms: neural networks, decision trees, and support vector machines. Each
algorithm was tested on datasets ranging from 1,000 to 1,000,000 records."
    ]
})

# Step 1: Split documents into manageable chunks
chunks = df.semantic.split(
    split_key="content",
    method="delimiter",
    method_kwargs={"delimiter": "\n\n", "num_splits_to_group": 1}
)

print(f"Original documents: {len(df)}")
print(f"Generated chunks: {len(chunks)}")

# Step 2: Add contextual information to each chunk
enhanced_chunks = chunks.semantic.gather(
    content_key="content_chunk",
    doc_id_key="semantic_split_0_id",
    order_key="semantic_split_0_chunk_num",
    peripheral_chunks={
        "previous": {"head": {"count": 1}},  # Include 1 previous chunk
        "next": {"head": {"count": 1}}       # Include 1 next chunk
    }
)

# Step 3: Extract structured information from each chunk with context
analyzed_chunks = enhanced_chunks.semantic.map(
    prompt="""Analyze this document chunk with its surrounding context and
extract:
    1. Main topic or section
```

```
    2. Key concepts discussed
    3. Action items or requirements (if any)

    Document chunk with context:
    {{input.content_chunk_rendered}}""",
    output={
        "schema": {
            "section_topic": "str",
            "key_concepts": "list[str]",
            "action_items": "list[str]"
        }
    }
)

# Step 4: Aggregate insights by document
document_summaries = analyzed_chunks.semantic.agg(
    reduce_keys=["document_id"],
    reduce_prompt="""Create a comprehensive summary of this document based on
all its chunks:

    Document chunks:
    {% for chunk in inputs %}
    Section: {{chunk.section_topic}}
    Key concepts: {{chunk.key_concepts | join(', ')}}
    Action items: {{chunk.action_items | join(', ')}}
    ---
    {% endfor %}""",
    output={
        "schema": {
            "document_summary": "str",
            "all_key_concepts": "list[str]",
            "all_action_items": "list[str]"
        }
    }
)

print(f"Final document summaries: {len(document_summaries)}")
print(f"Total processing cost: ${document_summaries.semantic.total_cost}")
```

Example 9: Data Structure Processing with Unnest

Handle complex nested data structures commonly found in JSON APIs or survey responses:

```
# Survey data with nested responses
survey_df = pd.DataFrame({
    "respondent_id": [1, 2, 3],
    "demographics": [
        {"age": 25, "location": "NYC", "education": "Bachelor's"},
        {"age": 34, "location": "SF", "education": "Master's"},
        {"age": 28, "location": "Chicago", "education": "PhD"}
    ],
    "interests": [
        ["technology", "sports", "music"],
```

```
        ["science", "reading"],
        ["art", "travel", "cooking", "photography"]
    ],
    "ratings": [
        {"product_quality": 4, "customer_service": 5, "value": 3},
        {"product_quality": 5, "customer_service": 4, "value": 4},
        {"product_quality": 3, "customer_service": 3, "value": 5}
    ]
})

# Step 1: Unnest demographics into separate columns
with_demographics = survey_df.semantic.unnest(
    unnest_key="demographics",
    expand_fields=["age", "location", "education"]
)

# Step 2: Unnest interests (each interest becomes a separate row)
individual_interests = with_demographics.semantic.unnest(
    unnest_key="interests"
)

# Step 3: For ratings, expand all fields
with_ratings = individual_interests.semantic.unnest(
    unnest_key="ratings",
    expand_fields=["product_quality", "customer_service", "value"]
)

print(f"Original survey responses: {len(survey_df)}")
print(f"Individual interest entries: {len(individual_interests)}")
print(f"Final flattened dataset: {len(with_ratings)}")

# Now you can analyze individual interests by demographics
interest_analysis = with_ratings.semantic.map(
    prompt="""Analyze this person's interest in the context of their
demographics:

    Person: {{input.age}} years old, {{input.education}}, from
{{input.location}}
    Interest: {{input.interests}}
    Product ratings: Quality={{input.product_quality}}, Service=
{{input.customer_service}}, Value={{input.value}}

    Provide insights about how this interest might relate to their
demographics and satisfaction.""",
    output={
        "schema": {
            "demographic_insight": "str",
            "interest_category": "str",
            "satisfaction_correlation": "str"
        }
    }
)

# Aggregate insights by interest category
category_insights = interest_analysis.semantic.agg(
    reduce_keys=["interest_category"],
    reduce_prompt="""Summarize insights about people interested in
```

```
{{inputs[0].interest_category}}:

    {% for person in inputs %}
    - {{person.age}} year old {{person.education}} from {{person.location}}:
{{person.demographic_insight}}
    {% endfor %}""",
    output={
        "schema": {
            "category_summary": "str",
            "typical_demographics": "str",
            "satisfaction_patterns": "str"
        }
    }
)
```

Example 10: Combined Document Workflow

A comprehensive example combining multiple operations for end-to-end document processing:

```
# Research papers with metadata
papers_df = pd.DataFrame({
    "paper_id": ["P001", "P002", "P003"],
    "title": [
        "Deep Learning for Natural Language Processing",
        "Quantum Computing Applications in Cryptography",
        "Sustainable Energy Storage Solutions"
    ],
    "abstract": ["This paper explores...", "We investigate...", "This study
examines..."],
    "full_text": [
        "Introduction\n\nDeep learning has revolutionized
NLP...\n\nMethodology\n\nWe employed transformer
architectures...\n\nResults\n\nOur experiments show significant
improvements...\n\nConclusion\n\nThe findings demonstrate...",
        "Introduction\n\nQuantum computing
promises...\n\nBackground\n\nClassical cryptography relies...\n\nQuantum
Algorithms\n\nShor's algorithm can factor...\n\nConclusion\n\nQuantum
computing will require...",
        "Introduction\n\nRenewable energy adoption...\n\nCurrent
Challenges\n\nEnergy storage remains...\n\nProposed Solutions\n\nWe propose
novel battery...\n\nResults\n\nOur testing shows..."
    ],
    "authors": [
        ["Dr. Smith", "Prof. Johnson", "Dr. Lee"],
        ["Prof. Chen", "Dr. Wilson"],
        ["Dr. Brown", "Prof. Davis", "Dr. Taylor", "Prof. Anderson"]
    ]
})

# Step 1: Unnest authors for author-level analysis
author_papers = papers_df.semantic.unnest(unnest_key="authors")

# Step 2: Split full text into sections
```

```
paper_sections = papers_df.semantic.split(
    split_key="full_text",
    method="delimiter",
    method_kwargs={"delimiter": "\n\n", "num_splits_to_group": 1}
)

# Step 3: Add context to each section
contextual_sections = paper_sections.semantic.gather(
    content_key="full_text_chunk",
    doc_id_key="semantic_split_0_id",
    order_key="semantic_split_0_chunk_num",
    peripheral_chunks={
        "previous": {"head": {"count": 1}},
        "next": {"head": {"count": 1}}
    }
)

# Step 4: Extract insights from each section
section_insights = contextual_sections.semantic.map(
    prompt="""Analyze this paper section in context:

Paper: {{input.title}}
Section content: {{input.full_text_chunk_rendered}}

Extract:
1. Section type (Introduction, Methodology, Results, etc.)
2. Key findings or claims
3. Technical concepts mentioned
4. Research gaps or future work mentioned""",
    output={
        "schema": {
            "section_type": "str",
            "key_findings": "list[str]",
            "technical_concepts": "list[str]",
            "future_work": "list[str]"
        }
    }
)

# Step 5: Aggregate insights by paper
paper_summaries = section_insights.semantic.agg(
    reduce_keys=["paper_id"],
    reduce_prompt="""Create a comprehensive analysis of this research paper:

Paper: {{inputs[0].title}}

Sections analyzed:
{% for section in inputs %}
{{section.section_type}}: {{section.key_findings | join(', ')}}
Technical concepts: {{section.technical_concepts | join(', ')}}
{% endfor %}

Provide a structured summary.""",
    output={
        "schema": {
            "comprehensive_summary": "str",
            "main_contributions": "list[str]",
```

```
            "methodology_type": "str",
            "research_field": "str"
        }
    }
)

# Step 6: Cross-paper analysis
field_analysis = paper_summaries.semantic.agg(
    reduce_keys=["research_field"],
    fuzzy=True,  # Group similar research fields
    reduce_prompt="""Analyze research trends in this field based on these
papers:

{% for paper in inputs %}
Paper: {{paper.title}}
Summary: {{paper.comprehensive_summary}}
Contributions: {{paper.main_contributions | join(', ')}}
---
{% endfor %}""",
    output={
        "schema": {
            "field_trends": "str",
            "common_methodologies": "list[str]",
            "emerging_themes": "list[str]"
        }
    }
)

print(f"Papers processed: {len(papers_df)}")
print(f"Authors identified: {len(author_papers)}")
print(f"Sections analyzed: {len(section_insights)}")
print(f"Research fields identified: {len(field_analysis)}")
print(f"Total processing cost: ${field_analysis.semantic.total_cost}")
```

DocETL Optimizer

The DocETL optimizer finds a plan that improves the accuracy of your document processing pipelines. It works by analyzing and potentially rewriting operations marked for optimization, finding optimal plans for execution.

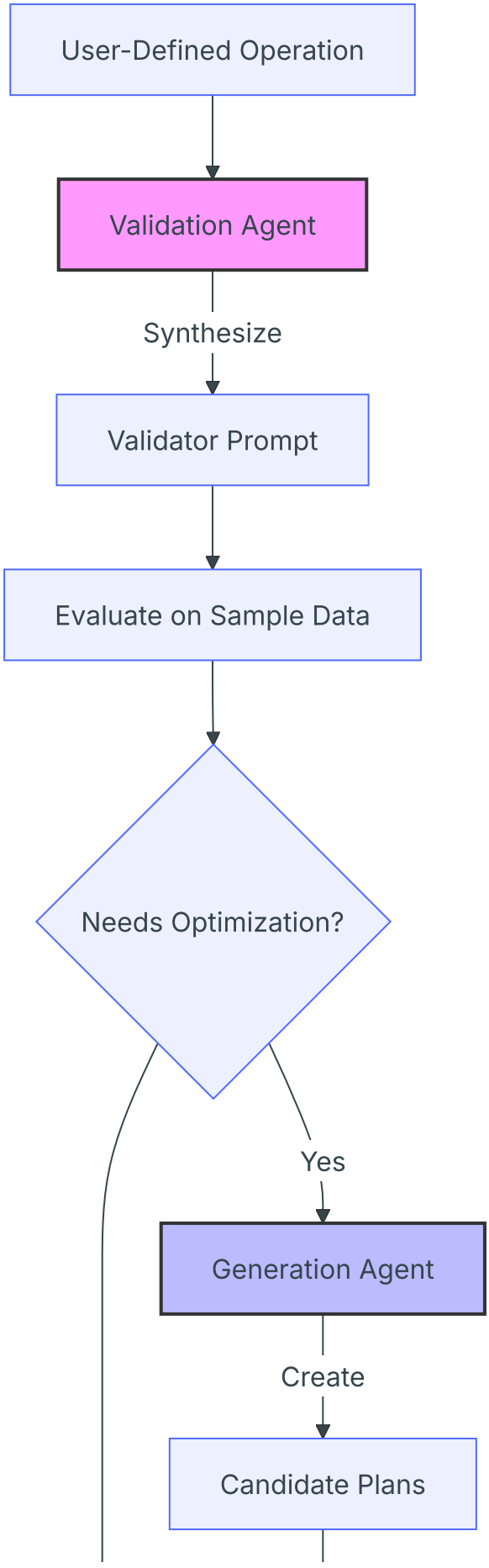
Key Features

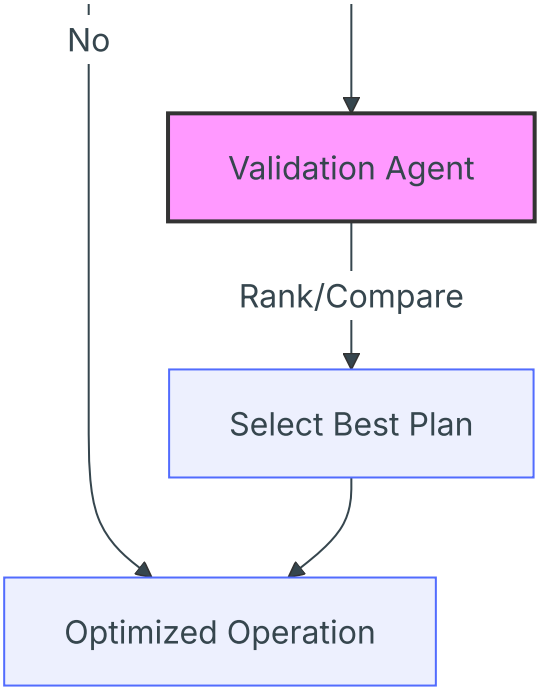
- Automatically decomposes complex operations into more efficient sub-pipelines
- Inserts resolve operations before reduce operations when beneficial
- Optimizes for large documents that exceed context limits
- Improves accuracy in high-volume reduce operations with incremental reduce

How It Works

The optimizer employs AI agents to generate and validate potential optimizations:

1. **Generation Agents**: Create alternative plans for operations, potentially breaking them down into multiple steps.
2. **Validation Agents**: Evaluate and compare the outputs of different plans to determine the most effective approach.


User-Defined Operation
Validation Agent
Synthesize
Validator Prompt
Evaluate on Sample Data
Needs Optimization?
Yes
Generation Agent
Create
Candidate Plans



Should I Use the Optimizer?

While any pipeline can potentially benefit from optimization, there are specific scenarios where using the optimizer can significantly improve your pipeline's performance and accuracy. When should you use the optimizer?

Large Documents

If you have documents that approach or exceed context limits and a map operation that transforms these documents using an LLM, the optimizer can help:

- Improve accuracy
- Enable processing of entire documents
- Optimize for large-scale data handling

Entity Resolution

The optimizer is particularly useful when:

```
- You need a resolve operation before your reduce operation
- You've defined a resolve operation but want to optimize it for speed using
blocking
```

High-Volume Reduce Operations

Consider using the optimizer when:

```
- You have many documents feeding into a reduce operation for a given key
- You're concerned about the accuracy of the reduce operation due to high
volume
- You want to optimize for better accuracy in complex reductions
```

Even if your pipeline doesn't fall into these specific categories, optimization can still be beneficial. For example, the optimizer can enhance your operations by adding gleaning to an operation, which uses an LLM-powered validator to ensure operation correctness. Learn more about gleaning.

Example: Optimizing Legal Contract Analysis

Let's consider a pipeline for analyzing legal contracts, extracting clauses, and summarizing them by type. Initially, you might have a single map operation to extract and tag clauses, followed by a reduce operation to summarize them. However, this approach might not be accurate enough for long contracts.

Initial Pipeline

In the initial pipeline, you might have a single map operation that attempts to extract all clauses and tag them with their types in one go. This is followed by a reduce operation that summarizes the clauses by type. Maybe the reduce operation accurately summarizes the clauses in a single LLM call per clause type, but the map operation might not be able to accurately extract and tag the clauses in a single LLM call.

Optimized Pipeline

After applying the optimizer, your pipeline could be transformed into a more efficient and accurate sub-pipeline:

1. **Split Operation**: Breaks down *each* long contract into manageable chunks.
2. **Map Operation**: Processes each chunk to extract and tag clauses.
3. **Reduce Operation**: For each contract, combine the extracted and tagged clauses from each chunk.

The goal of the DocETL optimizer is to try many ways of rewriting your pipeline and then select the best one. This may take some time (20-30 minutes for very complex tasks and large documents). But the optimizer's ability to break down complex tasks into more manageable sub-steps can lead to more accurate and reliable results.

Advanced: Customizing Optimization

You can customize the optimization process for specific operations using the ``optimizer_config in your pipeline.

Global Configuration

The following options can be applied globally to all operations in your pipeline during optimization:

- `num_retries`: The number of times to retry optimizing if the LLM agent fails. Default is 1.
- `sample_sizes`: Override the default sample sizes for each operator type. Specify as a dictionary with operator types as keys and integer sample sizes as values.

Default sample sizes:

```
SAMPLE_SIZE_MAP = {
    "reduce": 40,
    "map": 5,
    "resolve": 100,
    "equijoin": 100,
    "filter": 5,
}
```

- `judge_agent_model`: Specify the model to use for the judge agent. Default is `gpt-4o-mini`.
- `rewrite_agent_model`: Specify the model to use for the rewrite agent. Default is `gpt-4o`.
- `litellm_kwargs`: Specify the litellm kwargs to use for the optimization. Default is `{}`.

Equijoin Configuration

- `target_recall`: Change the default target recall (default is 0.95).

Resolve Configuration

- `target_recall`: Specify the target recall for the resolve operation.

Reduce Configuration

- `synthesize_resolve`: Set to `False` if you definitely don't want a resolve operation synthesized or want to turn off this rewrite rule.

Map Configuration

- `force_chunking_plan`: Set to `True` if you want the the optimizer to force plan that breaks up the input documents into chunks.
- `plan_types`: Specify the plan types to consider for the map operation. The available plan types are:
- `chunk`: Breaks up the input documents into chunks (i.e., data decomposition).
- `proj_synthesis`: Synthesizes 1+ projections (i.e., task decomposition).
- `glean`: Synthesizes a glean plan (i.e., uses LLM as a judge to refine the output).

Example Configuration

Here's an example of how to use the `optimizer_config` in your pipeline:

```yaml
optimizer_config:
  rewrite_agent_model: gpt-4o-mini
  judge_agent_model: gpt-4o-mini
  litellm_kwargs:
    temperature: 0.5
  num_retries: 2
  sample_sizes:
    map: 10
    reduce: 50
  reduce:
    synthesize_resolve: false
  map:
    plan_types: # Considers all these plan types
      - chunk
      - proj_synthesis
      - glean

operations:
  - name: extract_medications
    type: map
    optimize: true
    recursively_optimize: true # Recursively optimize the map operation (i.e.,
optimize any new operations that are synthesized)
    # ... other configuration ...

  - name: summarize_prescriptions
    type: reduce
    optimize: true
```

```
    # ... other configuration ...
# ... rest of the pipeline configuration ...
```

This configuration will:

1. Retry optimization up to 2 times for each operation if the LLM agent fails.
2. Use custom sample sizes for map (10) and reduce (50) operations.
3. Prevent the synthesis of resolve operations for reduce operations.
4. Consider all plan types for map operations.

Running the Optimizer

Optimizer Stability

The optimization process can be unstable, as well as resource-intensive (we've seen it take up to 10 minutes to optimize a single operation, spending up to ~$50 in API costs for end-to-end pipelines). We recommend optimizing one operation at a time and retrying if necessary, as results may vary between runs. This approach also allows you to confidently verify that each optimized operation is performing as expected before moving on to the next.

See the API for more details on how to resume the optimizer from a failed run, by rerunning `docetl build pipeline.yaml --resume` (with the `--resume` flag).

Also, you can use gpt-4o-mini for cheaper optimizations (rather than the default gpt-4o), which you can do via `docetl build pipeline.yaml --model=gpt-4o-mini`.

To optimize your pipeline, start with your initial configuration and follow these steps:

1. Set `optimize: True` for the operation you want to optimize (start with the first operation, if you're not sure which one).
2. Run the optimizer using the command `docetl build pipeline.yaml`. This will generate an optimized version in `pipeline_opt.yaml`.
3. Review the optimized operation in `pipeline_opt.yaml`. If you're satisfied with the changes, copy the optimized operation back into your original `pipeline.yaml`.
4. Move on to the next LLM-powered operation and repeat steps 1-3.
5. Once all operations are optimized, your `pipeline.yaml` will contain the fully optimized pipeline.

When optimizing a resolve operation, the optimizer will also set blocking configurations and thresholds, saving you from manual configuration.

Feeling Ambitious?

You can run the optimizer on your entire pipeline by setting `optimize: True` for each operation you want to optimize. But sometimes the agent fails to find a better plan, and you'll need to manually intervene. We are exploring human-in-the-loop optimization, where the optimizer can ask for human feedback to improve its plans.

Example: Optimizing a Medical Transcripts Pipeline

Let's walk through optimizing a pipeline for extracting medication information from medical transcripts. We'll start with an initial pipeline and optimize it step by step.

Initial Pipeline

```yaml
datasets:
  transcripts:
    path: medical_transcripts.json
    type: file

default_model: gpt-4o-mini

operations:
  - name: extract_medications
    type: map
    optimize: true
    output:
      schema:
        medication: list[str]
    prompt: |
      Analyze the transcript: {{ input.src }}
      List all medications mentioned.

  - name: unnest_medications
    type: unnest
    unnest_key: medication

  - name: summarize_prescriptions
    type: reduce
    optimize: true
    reduce_key:
      - medication
    output:
      schema:
        side_effects: str
        uses: str
    prompt: |
      Summarize side effects and uses of {{ reduce_key }} from:
      {% for value in inputs %}
      Transcript {{ loop.index }}: {{ value.src }}
      {% endfor %}

pipeline:
  output:
    path: medication_summaries.json
    type: file
  steps:
    - input: transcripts
      name: medical_info_extraction
      operations:
        - extract_medications
```

```
        - unnest_medications
        - summarize_prescriptions
```

Optimization Steps

First, we'll optimize the `extract_medications` operation. Set `optimize: True` for this operation and run the optimizer. Review the changes and integrate them into your pipeline.

Then, optimize the `summarize_prescriptions` operation by setting `optimize: True` and running `docetl build pipeline.yaml` again. The optimizer may suggest adding a resolve operation at this point, and will automatically configure blocking and thresholds. After completing all steps, your optimized pipeline might look like this:

Optimized Pipeline

```yaml
datasets:
  transcripts:
    path: medical_transcripts.json
    type: file

default_model: gpt-4o-mini

operations:
  - name: extract_medications
    type: map
    output:
      schema:
        medication: list[str]
    prompt: |
      Analyze the transcript: {{ input.src }}
      List all medications mentioned.
    gleaning:
      num_rounds: 1
      validation_prompt: |
        Evaluate the extraction for completeness and accuracy:
        1. Are all medications, dosages, and symptoms from the transcript
included?
        2. Is the extracted information correct and relevant?

  - name: unnest_medications
    type: unnest
    unnest_key: medication

  - name: resolve_medications
    type: resolve
    blocking_keys:
      - medication
    blocking_threshold: 0.7
    comparison_prompt: |
      Compare medications:
      1: {{ input1.medication }}
```

```
      2: {{ input2.medication }}
      Are these the same or closely related?
    resolution_prompt: |
      Standardize the name for:
      {% for entry in inputs %}
      - {{ entry.medication }}
      {% endfor %}

  - 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:
      {% for value in inputs %}
      Transcript {{ loop.index }}: {{ value.src }}
      {% endfor %}
    fold_batch_size: 10
    fold_prompt: |
      Update the existing summary of side effects and uses for {{ reduce_key
}} based on the following additional transcripts:
      {% for value in inputs %}
      Transcript {{ loop.index }}: {{ value.src }}
      {% endfor %}

      Existing summary:
      Side effects: {{ output.side_effects }}
      Uses: {{ output.uses }}

      Provide an updated and comprehensive summary, incorporating both the
existing information and any new insights from the additional transcripts.

pipeline:
  output:
    path: medication_summaries.json
    type: file
  steps:
    - input: transcripts
      name: medical_info_extraction
      operations:
        - extract_medications
        - unnest_medications
        - resolve_medications
        - summarize_prescriptions
```

This optimized pipeline now includes improved prompts, a resolve operation, and additional output fields for more comprehensive medication information extraction.

Feedback Welcome

We're continually improving the optimizer. Your feedback on its performance and usability is invaluable. Please share your experiences and suggestions!

Optimizer API

`docetl.cli.build(yaml_file=typer.Argument(..., help='Path to the YAML file containing the pipeline configuration'), max_threads=typer.Option(None, help='Maximum number of threads to use for running operations'), resume=typer.Option(False, help='Resume optimization from a previous build that may have failed'), save_path=typer.Option(None, help='Path to save the optimized pipeline configuration'))`

Build and optimize the configuration specified in the YAML file. Any arguments passed here will override the values in the YAML file.

Parameters:

Name	Type	Description	Default
`yaml_file`	`Path`	Path to the YAML file containing the pipeline configuration.	`Argument(..., help='Path to the YAML file containing the pipeline configuration')`
`max_threads`	`int | None`	Maximum number of threads to use for running operations.	`Option(None, help='Maximum number of threads to use for running operations')`
`model`	`str`	Model to use for optimization. Defaults to "gpt-4o".	*required*
`resume`	`bool`	Whether to resume optimization from a previous run. Defaults to False.	`Option(False, help='Resume optimization from a`

Name	Type	Description	Default
			`previous build that may have failed')`
`save_path`	`Path`	Path to save the optimized pipeline configuration.	`Option(None, help='Path to save the optimized pipeline configuration')`

Source code in `docetl/cli.py`

```python
@app.command()
def build(
    yaml_file: Path = typer.Argument(
        ..., help="Path to the YAML file containing the pipeline configuration"
    ),
    max_threads: int | None = typer.Option(
        None, help="Maximum number of threads to use for running operations"
    ),
    resume: bool = typer.Option(
        False, help="Resume optimization from a previous build that may have failed"
    ),
    save_path: Path = typer.Option(
        None, help="Path to save the optimized pipeline configuration"
    ),
):
    """
    Build and optimize the configuration specified in the YAML file.
    Any arguments passed here will override the values in the YAML file.

    Args:
        yaml_file (Path): Path to the YAML file containing the pipeline configuration.
        max_threads (int | None): Maximum number of threads to use for running operations.
        model (str): Model to use for optimization. Defaults to "gpt-4o".
        resume (bool): Whether to resume optimization from a previous run. Defaults to False.
        save_path (Path): Path to save the optimized pipeline configuration.
    """
    # Get the current working directory (where the user called the command)
    cwd = os.getcwd()

    # Load .env file from the current working directory
    env_file = os.path.join(cwd, ".env")
    if os.path.exists(env_file):
        load_dotenv(env_file)

    runner = DSLRunner.from_yaml(str(yaml_file), max_threads=max_threads)
    runner.optimize(
        save=True,
        return_pipeline=False,
        resume=resume,
        save_path=save_path,
    )
```

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.