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