LLM-Powered Operators

`docetl.operations.map.MapOperation`

Bases: `BaseOperation`

Source code in `docetl/operations/map.py`

```python
class MapOperation(BaseOperation):
    class schema(BaseOperation.schema):
        type: str = "map"
        output: dict[str, Any] | None = None
        prompt: str | None = None
        model: str | None = None
        optimize: bool | None = None
        recursively_optimize: bool | None = None
        sample_size: int | None = None
        tools: list[dict[str, Any]] | None = (
            None  # FIXME: Why isn't this using the Tool data class so validation works automatically?
        )
        validation_rules: list[str] | None = Field(None, alias="validate")
        num_retries_on_validate_failure: int | None = None
        drop_keys: list[str] | None = None
        timeout: int | None = None
        enable_observability: bool = False
        batch_size: int | None = None
        clustering_method: str | None = None
        batch_prompt: str | None = None
        litellm_completion_kwargs: dict[str, Any] = {}
        pdf_url_key: str | None = None
        flush_partial_result: bool = False
        # Calibration parameters
        calibrate: bool = False
        num_calibration_docs: int = Field(10, gt=0)

        @field_validator("batch_prompt")
        def validate_batch_prompt(cls, v):
            if v is not None:
                try:
                    template = Template(v)
                    # Test render with a minimal inputs list to validate template
                    template.render(inputs=[{}])
                except Exception as e:
                    raise ValueError(
                        f"Invalid Jinja2 template in 'batch_prompt' or missing required 'inputs' variable: {str(e)}"
                    ) from e
            return v

        @field_validator("prompt")
        def validate_prompt(cls, v):
            if v is not None:
                try:
                    Template(v)
                except Exception as e:
                    raise ValueError(
                        f"Invalid Jinja2 template in 'prompt': {str(e)}"
                    ) from e
            return v

        @field_validator("tools")
        def validate_tools(cls, v):
```

```
            if v is not None:
                for tool in v:
                    try:
                        tool_obj = Tool(**tool)
                    except Exception:
                        raise TypeError("Tool must be a dictionary")

                    if not (tool_obj.code and tool_obj.function):
                        raise ValueError(
                            "Tool is missing required 'code' or
'function' key"
                        )

                    if not isinstance(tool_obj.function, ToolFunction):
                        raise TypeError("'function' in tool must be a
dictionary")

                    for key in ["name", "description", "parameters"]:
                        if not getattr(tool_obj.function, key):
                            raise ValueError(
                                f"Tool is missing required '{key}' in
'function'"
                            )
            return v

        @model_validator(mode="after")
        def validate_prompt_and_output_requirements(self):
            # If drop_keys is not specified, both prompt and output must
be present
            if not self.drop_keys:
                if not self.prompt or not self.output:
                    raise ValueError(
                        "If 'drop_keys' is not specified, both 'prompt'
and 'output' must be present in the configuration"
                    )

                if self.output and not self.output.get("schema"):
                    raise ValueError("Missing 'schema' in 'output'
configuration")

            return self

    def __init__(
        self,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)
        self.max_batch_size: int = self.config.get(
            "max_batch_size", kwargs.get("max_batch_size", None)
        )
        self.clustering_method = "random"

    def _generate_calibration_context(self, input_data: list[dict]) ->
str:
        """
        Generate calibration context by running the operation on a sample
of documents
        and using an LLM to suggest prompt improvements for consistency.

        Returns:
```

```
            str: Additional context to add to the original prompt
        """
        import random

        # Set seed for reproducibility
        random.seed(42)

        # Sample documents for calibration
        num_calibration_docs = min(
            self.config.get("num_calibration_docs", 10), len(input_data)
        )
        if num_calibration_docs == len(input_data):
            calibration_sample = input_data
        else:
            calibration_sample = random.sample(input_data,
num_calibration_docs)

        self.console.log(
            f"[bold blue]Running calibration on {num_calibration_docs}
documents...[/bold blue]"
        )

        # Temporarily disable calibration to avoid infinite recursion
        original_calibrate = self.config.get("calibrate", False)
        self.config["calibrate"] = False

        try:
            # Run the map operation on the calibration sample
            calibration_results, _ = self.execute(calibration_sample)

            # Prepare the calibration analysis prompt
            calibration_prompt = f"""
The following prompt was applied to sample documents to generate these
input-output pairs:

"{self.config["prompt"]}"

Sample inputs and their outputs:
"""

            for i, (input_doc, output_doc) in enumerate(
                zip(calibration_sample, calibration_results)
            ):
                calibration_prompt += f"\n--- Example {i+1} ---\n"
                calibration_prompt += f"Input: {input_doc}\n"
                calibration_prompt += f"Output: {output_doc}\n"

            calibration_prompt += """
Based on these examples, provide reference anchors that will be appended
to the prompt to help maintain consistency when processing all documents.

DO NOT provide generic advice. Instead, use specific examples from above
as calibration points.
Note that the outputs might be incorrect, because the user's prompt was
not calibrated or rich in the first place.
You can ignore the outputs if they are incorrect, and focus on the
diversity of the inputs.

Format as concrete reference points:
- "For reference, consider '[specific input text]' → [output] as a
baseline for [category/level]"
```

```
- "Documents similar to '[specific input text]' should be classified as
[output]"

Reference anchors:"""

            # Call LLM to get calibration suggestions
            messages = [{"role": "user", "content": calibration_prompt}]
            # Use a copy of the user-provided completion kwargs so we
don't mutate the original
            # and avoid hard-coding temperature to a value that may not
be supported by certain models.
            completion_kwargs =
dict(self.config.get("litellm_completion_kwargs", {}))
            # If the user did not explicitly specify a temperature, let
the model default handle it
            # to prevent incompatibility errors with providers that don't
support 0.0.
            # If a temperature is already provided, respect the user's
choice.

            llm_result = self.runner.api.call_llm(
                self.config.get("model", self.default_model),
                "calibration",
                messages,
                {"calibration_context": "string"},
                timeout_seconds=self.config.get("timeout", 120),

max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
                bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
                litellm_completion_kwargs=completion_kwargs,
                op_config=self.config,
            )

            # Parse the response
            if hasattr(llm_result, "response"):
                calibration_context = self.runner.api.parse_llm_response(
                    llm_result.response,
                    schema={"calibration_context": "string"},
                    manually_fix_errors=self.manually_fix_errors,
                )[0].get("calibration_context", "")
            else:
                calibration_context = ""

            return calibration_context

        finally:
            # Restore original calibration setting
            self.config["calibrate"] = original_calibrate

    def execute(self, input_data: list[dict]) -> tuple[list[dict],
float]:
        """
        Executes the map operation on the provided input data.

        Args:
            input_data (list[dict]): The input data to process.

        Returns:
            tuple[list[dict], float]: A tuple containing the processed
results and the total cost of the operation.
```

```
262
263         This method performs the following steps:
264         1. If calibration is enabled, runs calibration to improve prompt
265 consistency
266         2. If a prompt is specified, it processes each input item using
267 the specified prompt and LLM model
268         3. Applies gleaning if configured
269         4. Validates the output
270         5. If drop_keys is specified, it drops the specified keys from
271 each document
272         6. Aggregates results and calculates total cost
273
274         The method uses parallel processing to improve performance.
275         """
276         # Check if there's no prompt and only drop_keys
277         if "prompt" not in self.config and "drop_keys" in self.config:
278             # If only drop_keys is specified, simply drop the keys and
279 return
280             dropped_results = []
281             for item in input_data:
282                 new_item = {
283                     k: v for k, v in item.items() if k not in
284 self.config["drop_keys"]
285                 }
286                 dropped_results.append(new_item)
287             return dropped_results, 0.0  # Return the modified data with
288 no cost
289
290         # Generate calibration context if enabled
291         calibration_context = ""
292         if self.config.get("calibrate", False) and "prompt" in
293 self.config:
294             calibration_context =
295 self._generate_calibration_context(input_data)
296             if calibration_context:
297                 # Store original prompt for potential restoration
298                 self._original_prompt = self.config["prompt"]
299                 # Augment the prompt with calibration context
300                 self.config["prompt"] = (
301                     f"{self.config['prompt']}\n\n{calibration_context}"
302                 )
303                 self.console.log(
304                     f"[bold green]New map ({self.config['name']}) prompt
305 augmented with context on how to improve consistency:[/bold green]
306 {self.config['prompt']}"
307                 )
308             else:
309                 self.console.log(
310                     f"[bold yellow]Extra context on how to improve
311 consistency failed to generate for map ({self.config['name']});
312 continuing with prompt as is.[/bold yellow]"
313                 )
314
315         if self.status:
316             self.status.stop()
317
318         def _process_map_item(
319             item: dict, initial_result: dict | None = None
320         ) -> tuple[dict | None, float]:
321
322             prompt = strict_render(self.config["prompt"], {"input":
```

```
item})
            messages = [{"role": "user", "content": prompt}]
            if self.config.get("pdf_url_key", None):
                # Append the pdf to the prompt
                try:
                    pdf_url = item[self.config["pdf_url_key"]]
                except KeyError:
                    raise ValueError(
                        f"PDF URL key '{self.config['pdf_url_key']}' not
found in input data"
                    )

                # Download content
                if pdf_url.startswith("http"):
                    file_data = requests.get(pdf_url).content
                else:
                    with open(pdf_url, "rb") as f:
                        file_data = f.read()
                encoded_file = base64.b64encode(file_data).decode("utf-
8")
                base64_url = f"data:application/pdf;base64,
{encoded_file}"

                messages[0]["content"] = [
                    {"type": "image_url", "image_url": {"url":
base64_url}},
                    {"type": "text", "text": prompt},
                ]

            def validation_fn(response: dict[str, Any] | ModelResponse):
                structured_mode = (
                    self.config.get("output", {}).get("mode")
                    == OutputMode.STRUCTURED_OUTPUT.value
                )
                output = (
                    self.runner.api.parse_llm_response(
                        response,
                        schema=self.config["output"]["schema"],
                        tools=self.config.get("tools", None),
                        manually_fix_errors=self.manually_fix_errors,
                        use_structured_output=structured_mode,
                    )[0]
                    if isinstance(response, ModelResponse)
                    else response
                )
                # Check that the output has all the keys in the schema
                for key in self.config["output"]["schema"]:
                    if key not in output:
                        return output, False

                for key, value in item.items():
                    if key not in self.config["output"]["schema"]:
                        output[key] = value
                if self.runner.api.validate_output(self.config, output,
self.console):
                    return output, True
                return output, False

            if self.runner.is_cancelled:
                raise asyncio.CancelledError("Operation was cancelled")
            llm_result = self.runner.api.call_llm(
```

```
                self.config.get("model", self.default_model),
                "map",
                messages,
                self.config["output"]["schema"],
                tools=self.config.get("tools", None),
                scratchpad=None,
                timeout_seconds=self.config.get("timeout", 120),

max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
                validation_config=(
                    {
                        "num_retries":
self.num_retries_on_validate_failure,
                        "val_rule": self.config.get("validate", []),
                        "validation_fn": validation_fn,
                    }
                    if self.config.get("validate", None)
                    else None
                ),
                gleaning_config=self.config.get("gleaning", None),
                verbose=self.config.get("verbose", False),
                bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
                initial_result=initial_result,
                litellm_completion_kwargs=self.config.get(
                    "litellm_completion_kwargs", {}
                ),
                op_config=self.config,
            )

            if llm_result.validated:
                # Parse the response
                if isinstance(llm_result.response, ModelResponse):
                    structured_mode = (
                        self.config.get("output", {}).get("mode")
                        == OutputMode.STRUCTURED_OUTPUT.value
                    )
                    outputs = self.runner.api.parse_llm_response(
                        llm_result.response,
                        schema=self.config["output"]["schema"],
                        tools=self.config.get("tools", None),
                        manually_fix_errors=self.manually_fix_errors,
                        use_structured_output=structured_mode,
                    )
                else:
                    outputs = [llm_result.response]

                # Augment the output with the original item
                outputs = [{**item, **output} for output in outputs]
                if self.config.get("enable_observability", False):
                    for output in outputs:
                        output[f"_observability_{self.config['name']}"] =
{
                            "prompt": prompt
                        }
                return outputs, llm_result.total_cost

            return None, llm_result.total_cost

        # If there's a batch prompt, let's use that
        def _process_map_batch(items: list[dict]) -> tuple[list[dict],
```

```
float]:
            total_cost = 0
            if len(items) > 1 and self.config.get("batch_prompt", None):
                # Raise error if pdf_url_key is set
                if self.config.get("pdf_url_key", None):
                    raise ValueError("Batch prompts do not support PDF
URLs")

                batch_prompt = strict_render(
                    self.config["batch_prompt"], {"inputs": items}
                )

                # Issue the batch call
                llm_result = self.runner.api.call_llm_batch(
                    self.config.get("model", self.default_model),
                    "batch map",
                    [{"role": "user", "content": batch_prompt}],
                    self.config["output"]["schema"],
                    verbose=self.config.get("verbose", False),
                    timeout_seconds=self.config.get("timeout", 120),
                    max_retries_per_timeout=self.config.get(
                        "max_retries_per_timeout", 2
                    ),
                    bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
                    litellm_completion_kwargs=self.config.get(
                        "litellm_completion_kwargs", {}
                    ),
                )
                total_cost += llm_result.total_cost

                # Parse the LLM response
                structured_mode = (
                    self.config.get("output", {}).get("mode")
                    == OutputMode.STRUCTURED_OUTPUT.value
                )
                parsed_output = self.runner.api.parse_llm_response(
                    llm_result.response,
                    self.config["output"]["schema"],
                    use_structured_output=structured_mode,
                )[0].get("results", [])
                items_and_outputs = [
                    (item, parsed_output[idx] if idx < len(parsed_output)
else None)
                    for idx, item in enumerate(items)
                ]
            else:
                items_and_outputs = [(item, None) for item in items]

            # Run _process_map_item for each item
            all_results = []
            if len(items_and_outputs) > 1:
                with ThreadPoolExecutor(max_workers=self.max_batch_size)
as executor:
                    futures = [
                        executor.submit(
                            _process_map_item,
                            items_and_outputs[i][0],
                            items_and_outputs[i][1],
                        )
                        for i in range(len(items_and_outputs))
```

```
                    ]
                    for i in range(len(futures)):
                        try:
                            results, item_cost = futures[i].result()
                            if results is not None:
                                all_results.extend(results)
                            total_cost += item_cost
                        except Exception as e:
                            if self.config.get("skip_on_error", False):
                                self.console.log(
                                    f"[bold red]Error in map operation
{self.config['name']}, skipping item:[/bold red] {e}"
                                )
                                continue
                            else:
                                raise e
            else:
                try:
                    results, item_cost = _process_map_item(
                        items_and_outputs[0][0], items_and_outputs[0][1]
                    )
                    if results is not None:
                        all_results.extend(results)
                    total_cost += item_cost
                except Exception as e:
                    if self.config.get("skip_on_error", False):
                        self.console.log(
                            f"[bold red]Error in map operation
{self.config['name']}, skipping item:[/bold red] {e}"
                        )
                    else:
                        raise e

            return all_results, total_cost

        with ThreadPoolExecutor(max_workers=self.max_batch_size) as
executor:
            batch_size = self.max_batch_size if self.max_batch_size is
not None else 1
            futures = []
            for i in range(0, len(input_data), batch_size):
                batch = input_data[i : i + batch_size]
                futures.append(executor.submit(_process_map_batch,
batch))
            results = []
            total_cost = 0
            pbar = RichLoopBar(
                range(len(futures)),
                desc=f"Processing {self.config['name']} (map) on all
documents",
                console=self.console,
            )
            for batch_index in pbar:
                result_list, item_cost = futures[batch_index].result()
                if result_list:
                    if "drop_keys" in self.config:
                        result_list = [
                            {
                                k: v
                                for k, v in result.items()
                                if k not in self.config["drop_keys"]
```

```
                                }
                                for result in result_list
                            ]
                        results.extend(result_list)
                        # --- BEGIN: Flush partial checkpoint ---
                        if self.config.get("flush_partial_results", False):
                            op_name = self.config["name"]
                            self.runner._flush_partial_results(
                                op_name, batch_index, result_list
                            )
                        # --- END: Flush partial checkpoint ---
                    total_cost += item_cost

            if self.status:
                self.status.start()

            return results, total_cost
```

`execute(input_data)`

Executes the map operation on the provided input data.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	The input data to process.	*required*

Returns:

Type	Description
`tuple[list[dict], float]`	tuple[list[dict], float]: A tuple containing the processed results and the total cost of the operation.

This method performs the following steps: 1. If calibration is enabled, runs calibration to improve prompt consistency 2. If a prompt is specified, it processes each input item using the specified prompt and LLM model 3. Applies gleaning if configured 4. Validates the output 5. If drop_keys is specified, it drops the specified keys from each document 6. Aggregates results and calculates total cost

The method uses parallel processing to improve performance.

Source code in `docetl/operations/map.py`

```python
def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
    """
    Executes the map operation on the provided input data.

    Args:
        input_data (list[dict]): The input data to process.

    Returns:
        tuple[list[dict], float]: A tuple containing the processed
results and the total cost of the operation.

    This method performs the following steps:
    1. If calibration is enabled, runs calibration to improve prompt
consistency
    2. If a prompt is specified, it processes each input item using the
specified prompt and LLM model
    3. Applies gleaning if configured
    4. Validates the output
    5. If drop_keys is specified, it drops the specified keys from each
document
    6. Aggregates results and calculates total cost

    The method uses parallel processing to improve performance.
    """
    # Check if there's no prompt and only drop_keys
    if "prompt" not in self.config and "drop_keys" in self.config:
        # If only drop_keys is specified, simply drop the keys and return
        dropped_results = []
        for item in input_data:
            new_item = {
                k: v for k, v in item.items() if k not in
self.config["drop_keys"]
            }
            dropped_results.append(new_item)
        return dropped_results, 0.0  # Return the modified data with no
cost

    # Generate calibration context if enabled
    calibration_context = ""
    if self.config.get("calibrate", False) and "prompt" in self.config:
        calibration_context =
self._generate_calibration_context(input_data)
        if calibration_context:
            # Store original prompt for potential restoration
            self._original_prompt = self.config["prompt"]
            # Augment the prompt with calibration context
            self.config["prompt"] = (
                f"{self.config['prompt']}\n\n{calibration_context}"
            )
            self.console.log(
                f"[bold green]New map ({self.config['name']}) prompt
augmented with context on how to improve consistency:[/bold green]
{self.config['prompt']}"
            )
        else:
            self.console.log(
                f"[bold yellow]Extra context on how to improve
```

```
consistency failed to generate for map ({self.config['name']});
continuing with prompt as is.[/bold yellow]"
            )

    if self.status:
        self.status.stop()

    def _process_map_item(
        item: dict, initial_result: dict | None = None
    ) -> tuple[dict | None, float]:

        prompt = strict_render(self.config["prompt"], {"input": item})
        messages = [{"role": "user", "content": prompt}]
        if self.config.get("pdf_url_key", None):
            # Append the pdf to the prompt
            try:
                pdf_url = item[self.config["pdf_url_key"]]
            except KeyError:
                raise ValueError(
                    f"PDF URL key '{self.config['pdf_url_key']}' not
found in input data"
                )

            # Download content
            if pdf_url.startswith("http"):
                file_data = requests.get(pdf_url).content
            else:
                with open(pdf_url, "rb") as f:
                    file_data = f.read()
            encoded_file = base64.b64encode(file_data).decode("utf-8")
            base64_url = f"data:application/pdf;base64,{encoded_file}"

            messages[0]["content"] = [
                {"type": "image_url", "image_url": {"url": base64_url}},
                {"type": "text", "text": prompt},
            ]

        def validation_fn(response: dict[str, Any] | ModelResponse):
            structured_mode = (
                self.config.get("output", {}).get("mode")
                == OutputMode.STRUCTURED_OUTPUT.value
            )
            output = (
                self.runner.api.parse_llm_response(
                    response,
                    schema=self.config["output"]["schema"],
                    tools=self.config.get("tools", None),
                    manually_fix_errors=self.manually_fix_errors,
                    use_structured_output=structured_mode,
                )[0]
                if isinstance(response, ModelResponse)
                else response
            )
            # Check that the output has all the keys in the schema
            for key in self.config["output"]["schema"]:
                if key not in output:
                    return output, False

            for key, value in item.items():
                if key not in self.config["output"]["schema"]:
                    output[key] = value
```

```
            if self.runner.api.validate_output(self.config, output,
self.console):
                return output, True
            return output, False

        if self.runner.is_cancelled:
            raise asyncio.CancelledError("Operation was cancelled")
        llm_result = self.runner.api.call_llm(
            self.config.get("model", self.default_model),
            "map",
            messages,
            self.config["output"]["schema"],
            tools=self.config.get("tools", None),
            scratchpad=None,
            timeout_seconds=self.config.get("timeout", 120),

max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
            validation_config=(
                {
                    "num_retries": self.num_retries_on_validate_failure,
                    "val_rule": self.config.get("validate", []),
                    "validation_fn": validation_fn,
                }
                if self.config.get("validate", None)
                else None
            ),
            gleaning_config=self.config.get("gleaning", None),
            verbose=self.config.get("verbose", False),
            bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
            initial_result=initial_result,
            litellm_completion_kwargs=self.config.get(
                "litellm_completion_kwargs", {}
            ),
            op_config=self.config,
        )

        if llm_result.validated:
            # Parse the response
            if isinstance(llm_result.response, ModelResponse):
                structured_mode = (
                    self.config.get("output", {}).get("mode")
                    == OutputMode.STRUCTURED_OUTPUT.value
                )
                outputs = self.runner.api.parse_llm_response(
                    llm_result.response,
                    schema=self.config["output"]["schema"],
                    tools=self.config.get("tools", None),
                    manually_fix_errors=self.manually_fix_errors,
                    use_structured_output=structured_mode,
                )
            else:
                outputs = [llm_result.response]

            # Augment the output with the original item
            outputs = [{**item, **output} for output in outputs]
            if self.config.get("enable_observability", False):
                for output in outputs:
                    output[f"_observability_{self.config['name']}"] = {
                        "prompt": prompt
                    }
```

```
            return outputs, llm_result.total_cost

        return None, llm_result.total_cost

    # If there's a batch prompt, let's use that
    def _process_map_batch(items: list[dict]) -> tuple[list[dict],
float]:
        total_cost = 0
        if len(items) > 1 and self.config.get("batch_prompt", None):
            # Raise error if pdf_url_key is set
            if self.config.get("pdf_url_key", None):
                raise ValueError("Batch prompts do not support PDF URLs")

            batch_prompt = strict_render(
                self.config["batch_prompt"], {"inputs": items}
            )

            # Issue the batch call
            llm_result = self.runner.api.call_llm_batch(
                self.config.get("model", self.default_model),
                "batch map",
                [{"role": "user", "content": batch_prompt}],
                self.config["output"]["schema"],
                verbose=self.config.get("verbose", False),
                timeout_seconds=self.config.get("timeout", 120),
                max_retries_per_timeout=self.config.get(
                    "max_retries_per_timeout", 2
                ),
                bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
                litellm_completion_kwargs=self.config.get(
                    "litellm_completion_kwargs", {}
                ),
            )
            total_cost += llm_result.total_cost

            # Parse the LLM response
            structured_mode = (
                self.config.get("output", {}).get("mode")
                == OutputMode.STRUCTURED_OUTPUT.value
            )
            parsed_output = self.runner.api.parse_llm_response(
                llm_result.response,
                self.config["output"]["schema"],
                use_structured_output=structured_mode,
            )[0].get("results", [])
            items_and_outputs = [
                (item, parsed_output[idx] if idx < len(parsed_output)
else None)
                for idx, item in enumerate(items)
            ]
        else:
            items_and_outputs = [(item, None) for item in items]

        # Run _process_map_item for each item
        all_results = []
        if len(items_and_outputs) > 1:
            with ThreadPoolExecutor(max_workers=self.max_batch_size) as
executor:
                futures = [
                    executor.submit(
```

```
                        _process_map_item,
                        items_and_outputs[i][0],
                        items_and_outputs[i][1],
                    )
                    for i in range(len(items_and_outputs))
                ]
                for i in range(len(futures)):
                    try:
                        results, item_cost = futures[i].result()
                        if results is not None:
                            all_results.extend(results)
                        total_cost += item_cost
                    except Exception as e:
                        if self.config.get("skip_on_error", False):
                            self.console.log(
                                f"[bold red]Error in map operation
{self.config['name']}, skipping item:[/bold red] {e}"
                            )
                            continue
                        else:
                            raise e
        else:
            try:
                results, item_cost = _process_map_item(
                    items_and_outputs[0][0], items_and_outputs[0][1]
                )
                if results is not None:
                    all_results.extend(results)
                total_cost += item_cost
            except Exception as e:
                if self.config.get("skip_on_error", False):
                    self.console.log(
                        f"[bold red]Error in map operation
{self.config['name']}, skipping item:[/bold red] {e}"
                    )
                else:
                    raise e

        return all_results, total_cost

    with ThreadPoolExecutor(max_workers=self.max_batch_size) as executor:
        batch_size = self.max_batch_size if self.max_batch_size is not
None else 1
        futures = []
        for i in range(0, len(input_data), batch_size):
            batch = input_data[i : i + batch_size]
            futures.append(executor.submit(_process_map_batch, batch))
        results = []
        total_cost = 0
        pbar = RichLoopBar(
            range(len(futures)),
            desc=f"Processing {self.config['name']} (map) on all
documents",
            console=self.console,
        )
        for batch_index in pbar:
            result_list, item_cost = futures[batch_index].result()
            if result_list:
                if "drop_keys" in self.config:
                    result_list = [
                        {
```

```
                                    k: v
                                    for k, v in result.items()
                                    if k not in self.config["drop_keys"]
                                }
                                for result in result_list
                            ]
                        results.extend(result_list)
                        # --- BEGIN: Flush partial checkpoint ---
                        if self.config.get("flush_partial_results", False):
                            op_name = self.config["name"]
                            self.runner._flush_partial_results(
                                op_name, batch_index, result_list
                            )
                        # --- END: Flush partial checkpoint ---
                    total_cost += item_cost

            if self.status:
                self.status.start()

            return results, total_cost
```

`docetl.operations.resolve.ResolveOperation`

Bases: `BaseOperation`

Source code in `docetl/operations/resolve.py`

```python
class ResolveOperation(BaseOperation):
    class schema(BaseOperation.schema):
        type: str = "resolve"
        comparison_prompt: str
        resolution_prompt: str | None = None
        output: dict[str, Any] | None = None
        embedding_model: str | None = None
        resolution_model: str | None = None
        comparison_model: str | None = None
        blocking_keys: list[str] | None = None
        blocking_threshold: float | None = Field(None, ge=0, le=1)
        blocking_conditions: list[str] | None = None
        input: dict[str, Any] | None = None
        embedding_batch_size: int | None = Field(None, gt=0)
        compare_batch_size: int | None = Field(None, gt=0)
        limit_comparisons: int | None = Field(None, gt=0)
        optimize: bool | None = None
        timeout: int | None = Field(None, gt=0)
        litellm_completion_kwargs: dict[str, Any] =
Field(default_factory=dict)
        enable_observability: bool = False

        @field_validator("comparison_prompt")
        def validate_comparison_prompt(cls, v):
            if v is not None:
                try:
                    comparison_template = Template(v)
                    comparison_vars =
comparison_template.environment.parse(v).find_all(
                        jinja2.nodes.Name
                    )
                    comparison_var_names = {var.name for var in
comparison_vars}
                    if (
                        "input1" not in comparison_var_names
                        or "input2" not in comparison_var_names
                    ):
                        raise ValueError(
                            f"'comparison_prompt' must contain both
'input1' and 'input2' variables. {v}"
                        )
                except Exception as e:
                    raise ValueError(
                        f"Invalid Jinja2 template in 'comparison_prompt':
{str(e)}"
                    )
            return v

        @field_validator("resolution_prompt")
        def validate_resolution_prompt(cls, v):
            if v is not None:
                try:
                    reduction_template = Template(v)
                    reduction_vars =
reduction_template.environment.parse(v).find_all(
                        jinja2.nodes.Name
                    )
```

```
                    reduction_var_names = {var.name for var in
reduction_vars}
                    if "inputs" not in reduction_var_names:
                        raise ValueError(
                            "'resolution_prompt' must contain 'inputs'
variable"
                        )
                except Exception as e:
                    raise ValueError(
                        f"Invalid Jinja2 template in 'resolution_prompt':
{str(e)}"
                    )
            return v

        @field_validator("input")
        def validate_input_schema(cls, v):
            if v is not None:
                if "schema" not in v:
                    raise ValueError("Missing 'schema' in 'input'
configuration")
                if not isinstance(v["schema"], dict):
                    raise TypeError(
                        "'schema' in 'input' configuration must be a
dictionary"
                    )
            return v

        @model_validator(mode="after")
        def validate_output_schema(self, info: ValidationInfo):
            # Skip validation if we're using from dataframe accessors
            if isinstance(info.context, dict) and info.context.get(
                "_from_df_accessors"
            ):
                return self

            if self.output is None:
                raise ValueError(
                    "Missing required key 'output' in ResolveOperation
configuration"
                )

            if "schema" not in self.output:
                raise ValueError("Missing 'schema' in 'output'
configuration")

            if not isinstance(self.output["schema"], dict):
                raise TypeError(
                    "'schema' in 'output' configuration must be a
dictionary"
                )

            if not self.output["schema"]:
                raise ValueError("'schema' in 'output' configuration
cannot be empty")

            return self

    def compare_pair(
        self,
        comparison_prompt: str,
        model: str,
```

```
        item1: dict,
        item2: dict,
        blocking_keys: list[str] = [],
        timeout_seconds: int = 120,
        max_retries_per_timeout: int = 2,
    ) -> tuple[bool, float, str]:
        """
        Compares two items using an LLM model to determine if they match.

        Args:
            comparison_prompt (str): The prompt template for comparison.
            model (str): The LLM model to use for comparison.
            item1 (dict): The first item to compare.
            item2 (dict): The second item to compare.

        Returns:
            tuple[bool, float, str]: A tuple containing a boolean
indicating whether the items match, the cost of the comparison, and the
prompt.
        """
        if blocking_keys:
            if all(
                key in item1
                and key in item2
                and str(item1[key]).lower() == str(item2[key]).lower()
                for key in blocking_keys
            ):
                return True, 0, ""

        prompt = strict_render(comparison_prompt, {"input1": item1,
"input2": item2})
        response = self.runner.api.call_llm(
            model,
            "compare",
            [{"role": "user", "content": prompt}],
            {"is_match": "bool"},
            timeout_seconds=timeout_seconds,
            max_retries_per_timeout=max_retries_per_timeout,
            bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),

litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
{}),
            op_config=self.config,
        )
        output = self.runner.api.parse_llm_response(
            response.response,
            {"is_match": "bool"},
        )[0]

        return output["is_match"], response.total_cost, prompt

    def syntax_check(self) -> None:
        context = {"_from_df_accessors": self.runner._from_df_accessors}
        super().syntax_check(context)

    def validation_fn(self, response: dict[str, Any]):
        output = self.runner.api.parse_llm_response(
            response,
            schema=self.config["output"]["schema"],
        )[0]
```

```
        if self.runner.api.validate_output(self.config, output,
self.console):
            return output, True
        return output, False

    def execute(self, input_data: list[dict]) -> tuple[list[dict],
float]:
        """
        Executes the resolve operation on the provided dataset.

        Args:
            input_data (list[dict]): The dataset to resolve.

        Returns:
            tuple[list[dict], float]: A tuple containing the resolved
results and the total cost of the operation.

        This method performs the following steps:
        1. Initial blocking based on specified conditions and/or
embedding similarity
        2. Pairwise comparison of potentially matching entries using LLM
        3. Clustering of matched entries
        4. Resolution of each cluster into a single entry (if applicable)
        5. Result aggregation and validation

        The method also calculates and logs statistics such as
comparisons saved by blocking and self-join selectivity.
        """
        if len(input_data) == 0:
            return [], 0

        # Initialize observability data for all items at the start
        if self.config.get("enable_observability", False):
            observability_key = f"_observability_{self.config['name']}"
            for item in input_data:
                if observability_key not in item:
                    item[observability_key] = {
                        "comparison_prompts": [],
                        "resolution_prompt": None,
                    }

        blocking_keys = self.config.get("blocking_keys", [])
        blocking_threshold = self.config.get("blocking_threshold")
        blocking_conditions = self.config.get("blocking_conditions", [])
        if self.status:
            self.status.stop()

        if not blocking_threshold and not blocking_conditions:
            # Prompt the user for confirmation
            if not Confirm.ask(
                "[yellow]Warning: No blocking keys or conditions
specified. "
                "This may result in a large number of comparisons. "
                "We recommend specifying at least one blocking key or
condition, or using the optimizer to automatically come up with these. "
                "Do you want to continue without blocking?[/yellow]",
                console=self.runner.console,
            ):
                raise ValueError("Operation cancelled by user.")

        input_schema = self.config.get("input", {}).get("schema", {})
```

```
        if not blocking_keys:
            # Set them to all keys in the input data
            blocking_keys = list(input_data[0].keys())
        limit_comparisons = self.config.get("limit_comparisons")
        total_cost = 0

        def is_match(item1: dict[str, Any], item2: dict[str, Any]) ->
bool:
            return any(
                eval(condition, {"input1": item1, "input2": item2})
                for condition in blocking_conditions
            )

        # Calculate embeddings if blocking_threshold is set
        embeddings = None
        if blocking_threshold is not None:

            def get_embeddings_batch(
                items: list[dict[str, Any]]
            ) -> list[tuple[list[float], float]]:
                embedding_model = self.config.get(
                    "embedding_model", "text-embedding-3-small"
                )
                model_input_context_length =
model_cost.get(embedding_model, {}).get(
                    "max_input_tokens", 8192
                )

                texts = [
                    " ".join(str(item[key]) for key in blocking_keys if
key in item)[
                        : model_input_context_length * 3
                    ]
                    for item in items
                ]

                response = self.runner.api.gen_embedding(
                    model=embedding_model, input=texts
                )
                return [
                    (data["embedding"], completion_cost(response))
                    for data in response["data"]
                ]

            embeddings = []
            costs = []
            with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
                for i in range(
                    0, len(input_data),
self.config.get("embedding_batch_size", 1000)
                ):
                    batch = input_data[
                        i : i + self.config.get("embedding_batch_size",
1000)
                    ]
                    batch_results =
list(executor.map(get_embeddings_batch, [batch]))

                    for result in batch_results:
                        embeddings.extend([r[0] for r in result])
```

```
                        costs.extend([r[1] for r in result])

                total_cost += sum(costs)

        # Generate all pairs to compare, ensuring no duplicate
comparisons
        def get_unique_comparison_pairs() -> (
            tuple[list[tuple[int, int]], dict[tuple[str, ...],
list[int]]]
        ):
            # Create a mapping of values to their indices
            value_to_indices: dict[tuple[str, ...], list[int]] = {}
            for i, item in enumerate(input_data):
                # Create a hashable key from the blocking keys
                key = tuple(str(item.get(k, "")) for k in blocking_keys)
                if key not in value_to_indices:
                    value_to_indices[key] = []
                value_to_indices[key].append(i)

            # Generate pairs for comparison, comparing each unique value
combination only once
            comparison_pairs = []
            keys = list(value_to_indices.keys())

            # First, handle comparisons between different values
            for i in range(len(keys)):
                for j in range(i + 1, len(keys)):
                    # Only need one comparison between different values
                    idx1 = value_to_indices[keys[i]][0]
                    idx2 = value_to_indices[keys[j]][0]
                    if idx1 < idx2:  # Maintain ordering to avoid
duplicates
                        comparison_pairs.append((idx1, idx2))

            return comparison_pairs, value_to_indices

        comparison_pairs, value_to_indices =
get_unique_comparison_pairs()

        # Filter pairs based on blocking conditions
        def meets_blocking_conditions(pair: tuple[int, int]) -> bool:
            i, j = pair
            return (
                is_match(input_data[i], input_data[j]) if
blocking_conditions else False
            )

        blocked_pairs = (
            list(filter(meets_blocking_conditions, comparison_pairs))
            if blocking_conditions
            else comparison_pairs
        )

        # Apply limit_comparisons to blocked pairs
        if limit_comparisons is not None and len(blocked_pairs) >
limit_comparisons:
            self.console.log(
                f"Randomly sampling {limit_comparisons} pairs out of
{len(blocked_pairs)} blocked pairs."
            )
            blocked_pairs = random.sample(blocked_pairs,
```

```
limit_comparisons)

        # Initialize clusters with all indices
        clusters = [{i} for i in range(len(input_data))]
        cluster_map = {i: i for i in range(len(input_data))}

        # If there are remaining comparisons, fill with highest cosine
similarities
        remaining_comparisons = (
            limit_comparisons - len(blocked_pairs)
            if limit_comparisons is not None
            else float("inf")
        )
        if remaining_comparisons > 0 and blocking_threshold is not None:
            # Compute cosine similarity for all pairs efficiently
            from sklearn.metrics.pairwise import cosine_similarity

            similarity_matrix = cosine_similarity(embeddings)

            cosine_pairs = []
            for i, j in comparison_pairs:
                if (i, j) not in blocked_pairs and find_cluster(
                    i, cluster_map
                ) != find_cluster(j, cluster_map):
                    similarity = similarity_matrix[i, j]
                    if similarity >= blocking_threshold:
                        cosine_pairs.append((i, j, similarity))

            if remaining_comparisons != float("inf"):
                cosine_pairs.sort(key=lambda x: x[2], reverse=True)
                additional_pairs = [
                    (i, j) for i, j, _ in cosine_pairs[:
int(remaining_comparisons)]
                ]
                blocked_pairs.extend(additional_pairs)
            else:
                blocked_pairs.extend((i, j) for i, j, _ in cosine_pairs)

        # Modified merge_clusters to handle all indices with the same
value

        def merge_clusters(item1: int, item2: int) -> None:
            root1, root2 = find_cluster(item1, cluster_map),
find_cluster(
                item2, cluster_map
            )
            if root1 != root2:
                if len(clusters[root1]) < len(clusters[root2]):
                    root1, root2 = root2, root1
                clusters[root1] |= clusters[root2]
                cluster_map[root2] = root1
                clusters[root2] = set()

                # Also merge all other indices that share the same values
                key1 = tuple(str(input_data[item1].get(k, "")) for k in
blocking_keys)
                key2 = tuple(str(input_data[item2].get(k, "")) for k in
blocking_keys)

                # Merge all indices with the same values
                for idx in value_to_indices.get(key1, []):
```

```
                    if idx != item1:
                        root_idx = find_cluster(idx, cluster_map)
                        if root_idx != root1:
                            clusters[root1] |= clusters[root_idx]
                            cluster_map[root_idx] = root1
                            clusters[root_idx] = set()

                for idx in value_to_indices.get(key2, []):
                    if idx != item2:
                        root_idx = find_cluster(idx, cluster_map)
                        if root_idx != root1:
                            clusters[root1] |= clusters[root_idx]
                            cluster_map[root_idx] = root1
                            clusters[root_idx] = set()

        # Calculate and print statistics
        total_possible_comparisons = len(input_data) * (len(input_data) -
1) // 2
        comparisons_made = len(blocked_pairs)
        comparisons_saved = total_possible_comparisons - comparisons_made
        self.console.log(
            f"[green]Comparisons saved by blocking: {comparisons_saved} "
            f"({(comparisons_saved / total_possible_comparisons) *
100:.2f}%)[/green]"
        )
        self.console.log(
            f"[blue]Number of pairs to compare: {len(blocked_pairs)}
[/blue]"
        )

        # Compute an auto-batch size based on the number of comparisons
        def auto_batch() -> int:
            # Maximum batch size limit for 4o-mini model
            M = 500

            n = len(input_data)
            m = len(blocked_pairs)

            # https://www.wolframalpha.com/input?i=k%28k-
1%29%2F2+%2B+%28n-k%29%28k-1%29+%3D+m%2C+solve+for+k
            # Two possible solutions for k:
            # k = -1/2 sqrt((1 - 2n)^2 - 8m) + n + 1/2
            # k = 1/2 (sqrt((1 - 2n)^2 - 8m) + 2n + 1)

            discriminant = (1 - 2 * n) ** 2 - 8 * m
            sqrt_discriminant = discriminant**0.5

            k1 = -0.5 * sqrt_discriminant + n + 0.5
            k2 = 0.5 * (sqrt_discriminant + 2 * n + 1)

            # Take the maximum viable solution
            k = max(k1, k2)
            return M if k < 0 else min(int(k), M)

        # Compare pairs and update clusters in real-time
        batch_size = self.config.get("compare_batch_size", auto_batch())
        self.console.log(f"Using compare batch size: {batch_size}")
        pair_costs = 0

        pbar = RichLoopBar(
            range(0, len(blocked_pairs), batch_size),
```

```
            desc=f"Processing batches of {batch_size} LLM comparisons",
            console=self.console,
        )
        last_processed = 0
        for i in pbar:
            batch_end = last_processed + batch_size
            batch = blocked_pairs[last_processed:batch_end]
            # Filter pairs for the initial batch
            better_batch = [
                pair
                for pair in batch
                if find_cluster(pair[0], cluster_map) == pair[0]
                and find_cluster(pair[1], cluster_map) == pair[1]
            ]

            # Expand better_batch if it doesn't reach batch_size
            while len(better_batch) < batch_size and batch_end <
len(blocked_pairs):
                # Move batch_end forward by batch_size to get more pairs
                next_end = batch_end + batch_size
                next_batch = blocked_pairs[batch_end:next_end]

                better_batch.extend(
                    pair
                    for pair in next_batch
                    if find_cluster(pair[0], cluster_map) == pair[0]
                    and find_cluster(pair[1], cluster_map) == pair[1]
                )

                # Update batch_end to prevent overlapping in the next
loop
                batch_end = next_end
            better_batch = better_batch[:batch_size]
            last_processed = batch_end
            with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
                future_to_pair = {
                    executor.submit(
                        self.compare_pair,
                        self.config["comparison_prompt"],
                        self.config.get("comparison_model",
self.default_model),
                        input_data[pair[0]],
                        input_data[pair[1]],
                        blocking_keys,
                        timeout_seconds=self.config.get("timeout", 120),
                        max_retries_per_timeout=self.config.get(
                            "max_retries_per_timeout", 2
                        ),
                    ): pair
                    for pair in better_batch
                }

                for future in as_completed(future_to_pair):
                    pair = future_to_pair[future]
                    is_match_result, cost, prompt = future.result()
                    pair_costs += cost
                    if is_match_result:
                        merge_clusters(pair[0], pair[1])

                    if self.config.get("enable_observability", False):
```

```
                        observability_key =
f"_observability_{self.config['name']}"
                        for idx in (pair[0], pair[1]):
                            if observability_key not in input_data[idx]:
                                input_data[idx][observability_key] = {
                                    "comparison_prompts": [],
                                    "resolution_prompt": None,
                                }
                            input_data[idx][observability_key][
                                "comparison_prompts"
                            ].append(prompt)

        total_cost += pair_costs

        # Collect final clusters
        final_clusters = [cluster for cluster in clusters if cluster]

        # Process each cluster
        results = []

        def process_cluster(cluster):
            if len(cluster) > 1:
                cluster_items = [input_data[i] for i in cluster]
                if input_schema:
                    cluster_items = [
                        {k: item[k] for k in input_schema.keys() if k in
item}
                        for item in cluster_items
                    ]

                resolution_prompt = strict_render(
                    self.config["resolution_prompt"], {"inputs":
cluster_items}
                )
                reduction_response = self.runner.api.call_llm(
                    self.config.get("resolution_model",
self.default_model),
                    "reduce",
                    [{"role": "user", "content": resolution_prompt}],
                    self.config["output"]["schema"],
                    timeout_seconds=self.config.get("timeout", 120),
                    max_retries_per_timeout=self.config.get(
                        "max_retries_per_timeout", 2
                    ),
                    bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
                    validation_config=(
                        {
                            "val_rule": self.config.get("validate", []),
                            "validation_fn": self.validation_fn,
                        }
                        if self.config.get("validate", None)
                        else None
                    ),
                    litellm_completion_kwargs=self.config.get(
                        "litellm_completion_kwargs", {}
                    ),
                    op_config=self.config,
                )
                reduction_cost = reduction_response.total_cost

```

```
                if self.config.get("enable_observability", False):
                    for item in [input_data[i] for i in cluster]:
                        observability_key =
f"_observability_{self.config['name']}"
                        if observability_key not in item:
                            item[observability_key] = {
                                "comparison_prompts": [],
                                "resolution_prompt": None,
                            }
                        item[observability_key]["resolution_prompt"] =
resolution_prompt

                if reduction_response.validated:
                    reduction_output =
self.runner.api.parse_llm_response(
                        reduction_response.response,
                        self.config["output"]["schema"],
                        manually_fix_errors=self.manually_fix_errors,
                    )[0]

                    # If the output is overwriting an existing key, we
want to save the kv pairs
                    keys_in_output = [
                        k
                        for k in set(reduction_output.keys())
                        if k in cluster_items[0].keys()
                    ]

                    return (
                        [
                            {
                                **item,

f"_kv_pairs_preresolve_{self.config['name']}": {
                                    k: item[k] for k in keys_in_output
                                },
                                **{
                                    k: reduction_output[k]
                                    for k in self.config["output"]
["schema"]
                                },
                            }
                            for item in [input_data[i] for i in cluster]
                        ],
                        reduction_cost,
                    )
                return [], reduction_cost
            else:
                # Set the output schema to be the keys found in the
compare_prompt
                compare_prompt_keys = extract_jinja_variables(
                    self.config["comparison_prompt"]
                )
                # Get the set of keys in the compare_prompt
                compare_prompt_keys = set(
                    [
                        k.replace("input1.", "")
                        for k in compare_prompt_keys
                        if "input1" in k
                    ]
                )
```

```
694
695                 # For each key in the output schema, find the most
696 similar key in the compare_prompt
697                 output_keys = set(self.config["output"]["schema"].keys())
698                 key_mapping = {}
699                 for output_key in output_keys:
700                     best_match = None
701                     best_score = 0
702                     for compare_key in compare_prompt_keys:
703                         score = sum(
704                             c1 == c2 for c1, c2 in zip(output_key,
705 compare_key)
706                         ) / max(len(output_key), len(compare_key))
707                         if score > best_score:
708                             best_score = score
                            best_match = compare_key
                    key_mapping[output_key] = best_match

                # Create the result dictionary using the key mapping
                result = input_data[list(cluster)[0]].copy()
                result[f"_kv_pairs_preresolve_{self.config['name']}"] = {
                    ok: result[ck] for ok, ck in key_mapping.items() if
    ck in result
                }
                for output_key, compare_key in key_mapping.items():
                    if compare_key in input_data[list(cluster)[0]]:
                        result[output_key] = input_data[list(cluster)[0]]
    [compare_key]
                    elif output_key in input_data[list(cluster)[0]]:
                        result[output_key] = input_data[list(cluster)[0]]
    [output_key]
                    else:
                        result[output_key] = None  # or some default
    value

                return [result], 0

        # Calculate the number of records before and clusters after
        num_records_before = len(input_data)
        num_clusters_after = len(final_clusters)
        self.console.log(f"Number of keys before resolution:
    {num_records_before}")
        self.console.log(
            f"Number of distinct keys after resolution:
    {num_clusters_after}"
        )

        # If no resolution prompt is provided, we can skip the resolution
    phase
        # And simply select the most common value for each key
        if not self.config.get("resolution_prompt", None):
            for cluster in final_clusters:
                if len(cluster) > 1:
                    for key in self.config["output"]["keys"]:
                        most_common_value = max(
                            set(input_data[i][key] for i in cluster),
                            key=lambda x: sum(
                                1 for i in cluster if input_data[i][key]
    == x
                            ),
                        )
```

```
                            for i in cluster:
                                input_data[i][key] = most_common_value
                results = input_data
            else:
                with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
                    futures = [
                        executor.submit(process_cluster, cluster)
                        for cluster in final_clusters
                    ]
                    for future in rich_as_completed(
                        futures,
                        total=len(futures),
                        desc="Determining resolved key for each group of
equivalent keys",
                        console=self.console,
                    ):
                        cluster_results, cluster_cost = future.result()
                        results.extend(cluster_results)
                        total_cost += cluster_cost

            total_pairs = len(input_data) * (len(input_data) - 1) // 2
            true_match_count = sum(
                len(cluster) * (len(cluster) - 1) // 2
                for cluster in final_clusters
                if len(cluster) > 1
            )
            true_match_selectivity = (
                true_match_count / total_pairs if total_pairs > 0 else 0
            )
            self.console.log(f"Self-join selectivity:
{true_match_selectivity:.4f}")

            if self.status:
                self.status.start()

            return results, total_cost
```

`compare_pair(comparison_prompt, model, item1, item2, blocking_keys=[], timeout_seconds=120, max_retries_per_timeout=2)`

Compares two items using an LLM model to determine if they match.

Parameters:

Name	Type	Description	Default
`comparison_prompt`	`str`	The prompt template for comparison.	*required*
`model`	`str`	The LLM model to use for comparison.	*required*

Name	Type	Description	Default
`item1`	`dict`	The first item to compare.	*required*
`item2`	`dict`	The second item to compare.	*required*

Returns:

Type	Description
`tuple[bool, float, str]`	tuple[bool, float, str]: A tuple containing a boolean indicating whether the items match, the cost of the comparison, and the prompt.

Source code in `docetl/operations/resolve.py`

```python
def compare_pair(
    self,
    comparison_prompt: str,
    model: str,
    item1: dict,
    item2: dict,
    blocking_keys: list[str] = [],
    timeout_seconds: int = 120,
    max_retries_per_timeout: int = 2,
) -> tuple[bool, float, str]:
    """
    Compares two items using an LLM model to determine if they match.

    Args:
        comparison_prompt (str): The prompt template for comparison.
        model (str): The LLM model to use for comparison.
        item1 (dict): The first item to compare.
        item2 (dict): The second item to compare.

    Returns:
        tuple[bool, float, str]: A tuple containing a boolean indicating
whether the items match, the cost of the comparison, and the prompt.
    """
    if blocking_keys:
        if all(
            key in item1
            and key in item2
            and str(item1[key]).lower() == str(item2[key]).lower()
            for key in blocking_keys
        ):
            return True, 0, ""

    prompt = strict_render(comparison_prompt, {"input1": item1, "input2":
item2})
    response = self.runner.api.call_llm(
        model,
        "compare",
        [{"role": "user", "content": prompt}],
        {"is_match": "bool"},
        timeout_seconds=timeout_seconds,
        max_retries_per_timeout=max_retries_per_timeout,
        bypass_cache=self.config.get("bypass_cache", self.bypass_cache),

litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
{}),
        op_config=self.config,
    )
    output = self.runner.api.parse_llm_response(
        response.response,
        {"is_match": "bool"},
    )[0]

    return output["is_match"], response.total_cost, prompt
```

`execute(input_data)`

Executes the resolve operation on the provided dataset.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	The dataset to resolve.	*required*

Returns:

Type	Description
`tuple[list[dict], float]`	tuple[list[dict], float]: A tuple containing the resolved results and the total cost of the operation.

This method performs the following steps: 1. Initial blocking based on specified conditions and/or embedding similarity 2. Pairwise comparison of potentially matching entries using LLM 3. Clustering of matched entries 4. Resolution of each cluster into a single entry (if applicable) 5. Result aggregation and validation

The method also calculates and logs statistics such as comparisons saved by blocking and self-join selectivity.

Source code in `docetl/operations/resolve.py`

```python
def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
    """
    Executes the resolve operation on the provided dataset.

    Args:
        input_data (list[dict]): The dataset to resolve.

    Returns:
        tuple[list[dict], float]: A tuple containing the resolved results
and the total cost of the operation.

    This method performs the following steps:
    1. Initial blocking based on specified conditions and/or embedding
similarity
    2. Pairwise comparison of potentially matching entries using LLM
    3. Clustering of matched entries
    4. Resolution of each cluster into a single entry (if applicable)
    5. Result aggregation and validation

    The method also calculates and logs statistics such as comparisons
saved by blocking and self-join selectivity.
    """
    if len(input_data) == 0:
        return [], 0

    # Initialize observability data for all items at the start
    if self.config.get("enable_observability", False):
        observability_key = f"_observability_{self.config['name']}"
        for item in input_data:
            if observability_key not in item:
                item[observability_key] = {
                    "comparison_prompts": [],
                    "resolution_prompt": None,
                }

    blocking_keys = self.config.get("blocking_keys", [])
    blocking_threshold = self.config.get("blocking_threshold")
    blocking_conditions = self.config.get("blocking_conditions", [])
    if self.status:
        self.status.stop()

    if not blocking_threshold and not blocking_conditions:
        # Prompt the user for confirmation
        if not Confirm.ask(
            "[yellow]Warning: No blocking keys or conditions specified. "
            "This may result in a large number of comparisons. "
            "We recommend specifying at least one blocking key or
condition, or using the optimizer to automatically come up with these. "
            "Do you want to continue without blocking?[/yellow]",
            console=self.runner.console,
        ):
            raise ValueError("Operation cancelled by user.")

    input_schema = self.config.get("input", {}).get("schema", {})
    if not blocking_keys:
        # Set them to all keys in the input data
        blocking_keys = list(input_data[0].keys())
```

```
    limit_comparisons = self.config.get("limit_comparisons")
    total_cost = 0

    def is_match(item1: dict[str, Any], item2: dict[str, Any]) -> bool:
        return any(
            eval(condition, {"input1": item1, "input2": item2})
            for condition in blocking_conditions
        )

    # Calculate embeddings if blocking_threshold is set
    embeddings = None
    if blocking_threshold is not None:

        def get_embeddings_batch(
            items: list[dict[str, Any]]
        ) -> list[tuple[list[float], float]]:
            embedding_model = self.config.get(
                "embedding_model", "text-embedding-3-small"
            )
            model_input_context_length = model_cost.get(embedding_model,
{}).get(
                "max_input_tokens", 8192
            )

            texts = [
                " ".join(str(item[key]) for key in blocking_keys if key
in item)[
                    : model_input_context_length * 3
                ]
                for item in items
            ]

            response = self.runner.api.gen_embedding(
                model=embedding_model, input=texts
            )
            return [
                (data["embedding"], completion_cost(response))
                for data in response["data"]
            ]

        embeddings = []
        costs = []
        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            for i in range(
                0, len(input_data),
self.config.get("embedding_batch_size", 1000)
            ):
                batch = input_data[
                    i : i + self.config.get("embedding_batch_size", 1000)
                ]
                batch_results = list(executor.map(get_embeddings_batch,
[batch]))

                for result in batch_results:
                    embeddings.extend([r[0] for r in result])
                    costs.extend([r[1] for r in result])

            total_cost += sum(costs)

    # Generate all pairs to compare, ensuring no duplicate comparisons
```

```
    def get_unique_comparison_pairs() -> (
        tuple[list[tuple[int, int]], dict[tuple[str, ...], list[int]]]
    ):
        # Create a mapping of values to their indices
        value_to_indices: dict[tuple[str, ...], list[int]] = {}
        for i, item in enumerate(input_data):
            # Create a hashable key from the blocking keys
            key = tuple(str(item.get(k, "")) for k in blocking_keys)
            if key not in value_to_indices:
                value_to_indices[key] = []
            value_to_indices[key].append(i)

        # Generate pairs for comparison, comparing each unique value
combination only once
        comparison_pairs = []
        keys = list(value_to_indices.keys())

        # First, handle comparisons between different values
        for i in range(len(keys)):
            for j in range(i + 1, len(keys)):
                # Only need one comparison between different values
                idx1 = value_to_indices[keys[i]][0]
                idx2 = value_to_indices[keys[j]][0]
                if idx1 < idx2:  # Maintain ordering to avoid duplicates
                    comparison_pairs.append((idx1, idx2))

        return comparison_pairs, value_to_indices

    comparison_pairs, value_to_indices = get_unique_comparison_pairs()

    # Filter pairs based on blocking conditions
    def meets_blocking_conditions(pair: tuple[int, int]) -> bool:
        i, j = pair
        return (
            is_match(input_data[i], input_data[j]) if blocking_conditions
else False
        )

    blocked_pairs = (
        list(filter(meets_blocking_conditions, comparison_pairs))
        if blocking_conditions
        else comparison_pairs
    )

    # Apply limit_comparisons to blocked pairs
    if limit_comparisons is not None and len(blocked_pairs) >
limit_comparisons:
        self.console.log(
            f"Randomly sampling {limit_comparisons} pairs out of
{len(blocked_pairs)} blocked pairs."
        )
        blocked_pairs = random.sample(blocked_pairs, limit_comparisons)

    # Initialize clusters with all indices
    clusters = [{i} for i in range(len(input_data))]
    cluster_map = {i: i for i in range(len(input_data))}

    # If there are remaining comparisons, fill with highest cosine
similarities
    remaining_comparisons = (
        limit_comparisons - len(blocked_pairs)
```

```
        if limit_comparisons is not None
        else float("inf")
    )
    if remaining_comparisons > 0 and blocking_threshold is not None:
        # Compute cosine similarity for all pairs efficiently
        from sklearn.metrics.pairwise import cosine_similarity

        similarity_matrix = cosine_similarity(embeddings)

        cosine_pairs = []
        for i, j in comparison_pairs:
            if (i, j) not in blocked_pairs and find_cluster(
                i, cluster_map
            ) != find_cluster(j, cluster_map):
                similarity = similarity_matrix[i, j]
                if similarity >= blocking_threshold:
                    cosine_pairs.append((i, j, similarity))

        if remaining_comparisons != float("inf"):
            cosine_pairs.sort(key=lambda x: x[2], reverse=True)
            additional_pairs = [
                (i, j) for i, j, _ in cosine_pairs[:
int(remaining_comparisons)]
            ]
            blocked_pairs.extend(additional_pairs)
        else:
            blocked_pairs.extend((i, j) for i, j, _ in cosine_pairs)

    # Modified merge_clusters to handle all indices with the same value

    def merge_clusters(item1: int, item2: int) -> None:
        root1, root2 = find_cluster(item1, cluster_map), find_cluster(
            item2, cluster_map
        )
        if root1 != root2:
            if len(clusters[root1]) < len(clusters[root2]):
                root1, root2 = root2, root1
            clusters[root1] |= clusters[root2]
            cluster_map[root2] = root1
            clusters[root2] = set()

            # Also merge all other indices that share the same values
            key1 = tuple(str(input_data[item1].get(k, "")) for k in
blocking_keys)
            key2 = tuple(str(input_data[item2].get(k, "")) for k in
blocking_keys)

            # Merge all indices with the same values
            for idx in value_to_indices.get(key1, []):
                if idx != item1:
                    root_idx = find_cluster(idx, cluster_map)
                    if root_idx != root1:
                        clusters[root1] |= clusters[root_idx]
                        cluster_map[root_idx] = root1
                        clusters[root_idx] = set()

            for idx in value_to_indices.get(key2, []):
                if idx != item2:
                    root_idx = find_cluster(idx, cluster_map)
                    if root_idx != root1:
                        clusters[root1] |= clusters[root_idx]
```

```
                        cluster_map[root_idx] = root1
                        clusters[root_idx] = set()

    # Calculate and print statistics
    total_possible_comparisons = len(input_data) * (len(input_data) - 1)
// 2
    comparisons_made = len(blocked_pairs)
    comparisons_saved = total_possible_comparisons - comparisons_made
    self.console.log(
        f"[green]Comparisons saved by blocking: {comparisons_saved} "
        f"({(comparisons_saved / total_possible_comparisons) * 100:.2f}%)
[/green]"
    )
    self.console.log(
        f"[blue]Number of pairs to compare: {len(blocked_pairs)}[/blue]"
    )

    # Compute an auto-batch size based on the number of comparisons
    def auto_batch() -> int:
        # Maximum batch size limit for 4o-mini model
        M = 500

        n = len(input_data)
        m = len(blocked_pairs)

        # https://www.wolframalpha.com/input?i=k%28k-1%29%2F2+%2B+%28n-
k%29%28k-1%29+%3D+m%2C+solve+for+k
        # Two possible solutions for k:
        # k = -1/2 sqrt((1 - 2n)^2 - 8m) + n + 1/2
        # k = 1/2 (sqrt((1 - 2n)^2 - 8m) + 2n + 1)

        discriminant = (1 - 2 * n) ** 2 - 8 * m
        sqrt_discriminant = discriminant**0.5

        k1 = -0.5 * sqrt_discriminant + n + 0.5
        k2 = 0.5 * (sqrt_discriminant + 2 * n + 1)

        # Take the maximum viable solution
        k = max(k1, k2)
        return M if k < 0 else min(int(k), M)

    # Compare pairs and update clusters in real-time
    batch_size = self.config.get("compare_batch_size", auto_batch())
    self.console.log(f"Using compare batch size: {batch_size}")
    pair_costs = 0

    pbar = RichLoopBar(
        range(0, len(blocked_pairs), batch_size),
        desc=f"Processing batches of {batch_size} LLM comparisons",
        console=self.console,
    )
    last_processed = 0
    for i in pbar:
        batch_end = last_processed + batch_size
        batch = blocked_pairs[last_processed:batch_end]
        # Filter pairs for the initial batch
        better_batch = [
            pair
            for pair in batch
            if find_cluster(pair[0], cluster_map) == pair[0]
            and find_cluster(pair[1], cluster_map) == pair[1]
```

```
        ]

        # Expand better_batch if it doesn't reach batch_size
        while len(better_batch) < batch_size and batch_end <
len(blocked_pairs):
            # Move batch_end forward by batch_size to get more pairs
            next_end = batch_end + batch_size
            next_batch = blocked_pairs[batch_end:next_end]

            better_batch.extend(
                pair
                for pair in next_batch
                if find_cluster(pair[0], cluster_map) == pair[0]
                and find_cluster(pair[1], cluster_map) == pair[1]
            )

            # Update batch_end to prevent overlapping in the next loop
            batch_end = next_end
        better_batch = better_batch[:batch_size]
        last_processed = batch_end
        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            future_to_pair = {
                executor.submit(
                    self.compare_pair,
                    self.config["comparison_prompt"],
                    self.config.get("comparison_model",
self.default_model),
                    input_data[pair[0]],
                    input_data[pair[1]],
                    blocking_keys,
                    timeout_seconds=self.config.get("timeout", 120),
                    max_retries_per_timeout=self.config.get(
                        "max_retries_per_timeout", 2
                    ),
                ): pair
                for pair in better_batch
            }

            for future in as_completed(future_to_pair):
                pair = future_to_pair[future]
                is_match_result, cost, prompt = future.result()
                pair_costs += cost
                if is_match_result:
                    merge_clusters(pair[0], pair[1])

                if self.config.get("enable_observability", False):
                    observability_key =
f"_observability_{self.config['name']}"
                    for idx in (pair[0], pair[1]):
                        if observability_key not in input_data[idx]:
                            input_data[idx][observability_key] = {
                                "comparison_prompts": [],
                                "resolution_prompt": None,
                            }
                        input_data[idx][observability_key][
                            "comparison_prompts"
                        ].append(prompt)

    total_cost += pair_costs

```

```
    # Collect final clusters
    final_clusters = [cluster for cluster in clusters if cluster]

    # Process each cluster
    results = []

    def process_cluster(cluster):
        if len(cluster) > 1:
            cluster_items = [input_data[i] for i in cluster]
            if input_schema:
                cluster_items = [
                    {k: item[k] for k in input_schema.keys() if k in
item}
                    for item in cluster_items
                ]

            resolution_prompt = strict_render(
                self.config["resolution_prompt"], {"inputs":
cluster_items}
            )
            reduction_response = self.runner.api.call_llm(
                self.config.get("resolution_model", self.default_model),
                "reduce",
                [{"role": "user", "content": resolution_prompt}],
                self.config["output"]["schema"],
                timeout_seconds=self.config.get("timeout", 120),
                max_retries_per_timeout=self.config.get(
                    "max_retries_per_timeout", 2
                ),
                bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
                validation_config=(
                    {
                        "val_rule": self.config.get("validate", []),
                        "validation_fn": self.validation_fn,
                    }
                    if self.config.get("validate", None)
                    else None
                ),
                litellm_completion_kwargs=self.config.get(
                    "litellm_completion_kwargs", {}
                ),
                op_config=self.config,
            )
            reduction_cost = reduction_response.total_cost

            if self.config.get("enable_observability", False):
                for item in [input_data[i] for i in cluster]:
                    observability_key =
f"_observability_{self.config['name']}"
                    if observability_key not in item:
                        item[observability_key] = {
                            "comparison_prompts": [],
                            "resolution_prompt": None,
                        }
                    item[observability_key]["resolution_prompt"] =
resolution_prompt

            if reduction_response.validated:
                reduction_output = self.runner.api.parse_llm_response(
                    reduction_response.response,
```

```
                    self.config["output"]["schema"],
                    manually_fix_errors=self.manually_fix_errors,
                )[0]

                # If the output is overwriting an existing key, we want
to save the kv pairs
                keys_in_output = [
                    k
                    for k in set(reduction_output.keys())
                    if k in cluster_items[0].keys()
                ]

                return (
                    [
                        {
                            **item,

f"_kv_pairs_preresolve_{self.config['name']}": {
                                k: item[k] for k in keys_in_output
                            },
                            **{
                                k: reduction_output[k]
                                for k in self.config["output"]["schema"]
                            },
                        }
                        for item in [input_data[i] for i in cluster]
                    ],
                    reduction_cost,
                )
            return [], reduction_cost
        else:
            # Set the output schema to be the keys found in the
compare_prompt
            compare_prompt_keys = extract_jinja_variables(
                self.config["comparison_prompt"]
            )
            # Get the set of keys in the compare_prompt
            compare_prompt_keys = set(
                [
                    k.replace("input1.", "")
                    for k in compare_prompt_keys
                    if "input1" in k
                ]
            )

            # For each key in the output schema, find the most similar
key in the compare_prompt
            output_keys = set(self.config["output"]["schema"].keys())
            key_mapping = {}
            for output_key in output_keys:
                best_match = None
                best_score = 0
                for compare_key in compare_prompt_keys:
                    score = sum(
                        c1 == c2 for c1, c2 in zip(output_key,
compare_key)
                    ) / max(len(output_key), len(compare_key))
                    if score > best_score:
                        best_score = score
                        best_match = compare_key
                key_mapping[output_key] = best_match
```

```
673
674             # Create the result dictionary using the key mapping
675             result = input_data[list(cluster)[0]].copy()
676             result[f"_kv_pairs_preresolve_{self.config['name']}"] = {
677                 ok: result[ck] for ok, ck in key_mapping.items() if ck in
678 result
679             }
680             for output_key, compare_key in key_mapping.items():
681                 if compare_key in input_data[list(cluster)[0]]:
682                     result[output_key] = input_data[list(cluster)[0]]
683 [compare_key]
684                 elif output_key in input_data[list(cluster)[0]]:
685                     result[output_key] = input_data[list(cluster)[0]]
686 [output_key]
687                 else:
688                     result[output_key] = None  # or some default value
689
690             return [result], 0
691
692     # Calculate the number of records before and clusters after
693     num_records_before = len(input_data)
694     num_clusters_after = len(final_clusters)
695     self.console.log(f"Number of keys before resolution:
696 {num_records_before}")
697     self.console.log(
698         f"Number of distinct keys after resolution: {num_clusters_after}"
699     )
700
701     # If no resolution prompt is provided, we can skip the resolution
702 phase
703     # And simply select the most common value for each key
704     if not self.config.get("resolution_prompt", None):
705         for cluster in final_clusters:
706             if len(cluster) > 1:
707                 for key in self.config["output"]["keys"]:
708                     most_common_value = max(
                            set(input_data[i][key] for i in cluster),
                            key=lambda x: sum(
                                1 for i in cluster if input_data[i][key] == x
                            ),
                        )
                        for i in cluster:
                            input_data[i][key] = most_common_value
        results = input_data
    else:
        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            futures = [
                executor.submit(process_cluster, cluster)
                for cluster in final_clusters
            ]
            for future in rich_as_completed(
                futures,
                total=len(futures),
                desc="Determining resolved key for each group of
equivalent keys",
                console=self.console,
            ):
                cluster_results, cluster_cost = future.result()
                results.extend(cluster_results)
                total_cost += cluster_cost
```

```
        total_pairs = len(input_data) * (len(input_data) - 1) // 2
        true_match_count = sum(
            len(cluster) * (len(cluster) - 1) // 2
            for cluster in final_clusters
            if len(cluster) > 1
        )
        true_match_selectivity = (
            true_match_count / total_pairs if total_pairs > 0 else 0
        )
        self.console.log(f"Self-join selectivity:
    {true_match_selectivity:.4f}")

        if self.status:
            self.status.start()

        return results, total_cost
```

`docetl.operations.reduce.ReduceOperation`

Bases: `BaseOperation`

A class that implements a reduce operation on input data using language models.

This class extends BaseOperation to provide functionality for reducing grouped data using various strategies including batch reduce, incremental reduce, and parallel fold and merge.

Source code in `docetl/operations/reduce.py`

```python
class ReduceOperation(BaseOperation):
    """
    A class that implements a reduce operation on input data using language models.

    This class extends BaseOperation to provide functionality for reducing grouped data
    using various strategies including batch reduce, incremental reduce, and parallel fold and merge.
    """

    class schema(BaseOperation.schema):
        type: str = "reduce"
        reduce_key: str | list[str]
        output: dict[str, Any]
        prompt: str
        optimize: bool | None = None
        synthesize_resolve: bool | None = None
        model: str | None = None
        input: dict[str, Any] | None = None
        pass_through: bool | None = None
        associative: bool | None = None
        fold_prompt: str | None = None
        fold_batch_size: int | None = Field(None, gt=0)
        merge_prompt: str | None = None
        merge_batch_size: int | None = Field(None, gt=0)
        value_sampling: dict[str, Any] | None = None
        verbose: bool | None = None
        timeout: int | None = None
        litellm_completion_kwargs: dict[str, Any] = Field(default_factory=dict)
        enable_observability: bool = False

        @field_validator("prompt")
        def validate_prompt(cls, v):
            if v is not None:
                try:
                    template = Template(v)
                    template_vars = template.environment.parse(v).find_all(
                        jinja2.nodes.Name
                    )
                    template_var_names = {var.name for var in template_vars}
                    if "inputs" not in template_var_names:
                        raise ValueError(
                            "Prompt template must include the 'inputs' variable"
                        )
                except Exception as e:
                    raise ValueError(f"Invalid Jinja2 template in 'prompt': {str(e)}")
            return v

        @field_validator("fold_prompt")
        def validate_fold_prompt(cls, v):
            if v is not None:
```

```
                try:
                    fold_template = Template(v)
                    fold_template_vars =
fold_template.environment.parse(v).find_all(
                        jinja2.nodes.Name
                    )
                    fold_template_var_names = {var.name for var in
fold_template_vars}
                    required_vars = {"inputs", "output"}
                    if not
required_vars.issubset(fold_template_var_names):
                        raise ValueError(
                            f"Fold template must include variables:
{required_vars}. Current template includes: {fold_template_var_names}"
                        )
                except Exception as e:
                    raise ValueError(
                        f"Invalid Jinja2 template in 'fold_prompt':
{str(e)}"
                    )
            return v

        @field_validator("merge_prompt")
        def validate_merge_prompt(cls, v):
            if v is not None:
                try:
                    merge_template = Template(v)
                    merge_template_vars =
merge_template.environment.parse(v).find_all(
                        jinja2.nodes.Name
                    )
                    merge_template_var_names = {var.name for var in
merge_template_vars}
                    if "outputs" not in merge_template_var_names:
                        raise ValueError(
                            "Merge template must include the 'outputs'
variable"
                        )
                except Exception as e:
                    raise ValueError(
                        f"Invalid Jinja2 template in 'merge_prompt':
{str(e)}"
                    )
            return v

        @field_validator("value_sampling")
        def validate_value_sampling(cls, v):
            if v is not None:
                if v["enabled"]:
                    if v["method"] not in ["random", "first_n",
"cluster", "sem_sim"]:
                        raise ValueError(
                            "Invalid 'method'. Must be 'random',
'first_n', 'cluster', or 'sem_sim'"
                        )

                    if v["method"] == "embedding":
                        if "embedding_model" not in v:
                            raise ValueError(
                                "'embedding_model' is required when using
embedding-based sampling"
```

```
                            )
                        if "embedding_keys" not in v:
                            raise ValueError(
                                "'embedding_keys' is required when using
embedding-based sampling"
                            )
            return v

        @model_validator(mode="after")
        def validate_complex_requirements(self):
            # Check dependencies between merge_prompt and fold_prompt
            if self.merge_prompt and not self.fold_prompt:
                raise ValueError(
                    "'fold_prompt' is required when 'merge_prompt' is
specified"
                )

            # Check batch size requirements
            if self.fold_prompt and not self.fold_batch_size:
                raise ValueError(
                    "'fold_batch_size' is required when 'fold_prompt' is
specified"
                )
            if self.merge_prompt and not self.merge_batch_size:
                raise ValueError(
                    "'merge_batch_size' is required when 'merge_prompt'
is specified"
                )

            return self

    def __init__(self, *args, **kwargs):
        """
        Initialize the ReduceOperation.

        Args:
            *args: Variable length argument list.
            **kwargs: Arbitrary keyword arguments.
        """
        super().__init__(*args, **kwargs)
        self.min_samples = 5
        self.max_samples = 1000
        self.fold_times = deque(maxlen=self.max_samples)
        self.merge_times = deque(maxlen=self.max_samples)
        self.lock = Lock()
        self.config["reduce_key"] = (
            [self.config["reduce_key"]]
            if isinstance(self.config["reduce_key"], str)
            else self.config["reduce_key"]
        )
        self.intermediates = {}
        self.lineage_keys = self.config.get("output", {}).get("lineage",
[])

    def execute(self, input_data: list[dict]) -> tuple[list[dict],
float]:
        """
        Execute the reduce operation on the provided input data.

        This method sorts and groups the input data by the reduce key(s),
then processes each group
```

```
        using either parallel fold and merge, incremental reduce, or
batch reduce strategies.

        Args:
            input_data (list[dict]): The input data to process.

        Returns:
            tuple[list[dict], float]: A tuple containing the processed
results and the total cost of the operation.
        """
        if self.config.get("gleaning", {}).get("validation_prompt",
None):
            self.console.log(
                f"Using gleaning with validation prompt:
{self.config.get('gleaning', {}).get('validation_prompt', '')}"
            )

        reduce_keys = self.config["reduce_key"]
        if isinstance(reduce_keys, str):
            reduce_keys = [reduce_keys]
        input_schema = self.config.get("input", {}).get("schema", {})

        if self.status:
            self.status.stop()

        # Check if we need to group everything into one group
        if reduce_keys == ["_all"] or reduce_keys == "_all":
            grouped_data = [("_all", input_data)]
        else:
            # Group the input data by the reduce key(s) while maintaining
original order
            def get_group_key(item):
                key_values = []
                for key in reduce_keys:
                    value = item[key]
                    # Special handling for list-type values
                    if isinstance(value, list):
                        key_values.append(
                            tuple(sorted(value))
                        )  # Convert list to sorted tuple
                    else:
                        key_values.append(value)
                return tuple(key_values)

            grouped_data = {}
            for item in input_data:
                key = get_group_key(item)
                if key not in grouped_data:
                    grouped_data[key] = []
                grouped_data[key].append(item)

            # Convert the grouped data to a list of tuples
            grouped_data = list(grouped_data.items())

        def process_group(
            key: tuple, group_elems: list[dict]
        ) -> tuple[dict | None, float]:
            if input_schema:
                group_list = [
                    {k: item[k] for k in input_schema.keys() if k in
item}
```

```
                    for item in group_elems
                ]
            else:
                group_list = group_elems

            total_cost = 0.0

            # Apply value sampling if enabled
            value_sampling = self.config.get("value_sampling", {})
            if value_sampling.get("enabled", False):
                sample_size = min(value_sampling["sample_size"],
len(group_list))
                method = value_sampling["method"]

                if method == "random":
                    group_sample = random.sample(group_list, sample_size)
                    group_sample.sort(key=lambda x: group_list.index(x))
                elif method == "first_n":
                    group_sample = group_list[:sample_size]
                elif method == "cluster":
                    group_sample, embedding_cost =
self._cluster_based_sampling(
                        group_list, value_sampling, sample_size
                    )
                    group_sample.sort(key=lambda x: group_list.index(x))
                    total_cost += embedding_cost
                elif method == "sem_sim":
                    group_sample, embedding_cost =
self._semantic_similarity_sampling(
                        key, group_list, value_sampling, sample_size
                    )
                    group_sample.sort(key=lambda x: group_list.index(x))
                    total_cost += embedding_cost

                group_list = group_sample

            # Only execute merge-based plans if associative = True
            if "merge_prompt" in self.config and
self.config.get("associative", True):
                result, prompts, cost =
self._parallel_fold_and_merge(key, group_list)
            elif self.config.get("fold_batch_size", None) and
self.config.get(
                "fold_batch_size"
            ) >= len(group_list):
                # If the fold batch size is greater than or equal to the
number of items in the group,
                # we can just run a single fold operation
                result, prompt, cost = self._batch_reduce(key,
group_list)
                prompts = [prompt]
            elif "fold_prompt" in self.config:
                result, prompts, cost = self._incremental_reduce(key,
group_list)
            else:
                result, prompt, cost = self._batch_reduce(key,
group_list)
                prompts = [prompt]

            total_cost += cost

```

```
            # Add the counts of items in the group to the result
            result[f"_counts_prereduce_{self.config['name']}"] =
len(group_elems)

            if self.config.get("enable_observability", False):
                # Add the _observability_{self.config['name']} key to the
result
                result[f"_observability_{self.config['name']}"] =
{"prompts": prompts}

            # Apply pass-through at the group level
            if (
                result is not None
                and self.config.get("pass_through", False)
                and group_elems
            ):
                for k, v in group_elems[0].items():
                    if k not in self.config["output"]["schema"] and k not
in result:
                        result[k] = v

            # Add lineage information
            if result is not None and self.lineage_keys:
                lineage = []
                for item in group_elems:
                    lineage_item = {
                        k: item.get(k) for k in self.lineage_keys if k in
item
                    }
                    if lineage_item:
                        lineage.append(lineage_item)
                result[f"{self.config['name']}_lineage"] = lineage

            return result, total_cost

        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            futures = [
                executor.submit(process_group, key, group)
                for key, group in grouped_data
            ]
            results = []
            total_cost = 0
            for future in rich_as_completed(
                futures,
                total=len(futures),
                desc=f"Processing {self.config['name']} (reduce) on all
documents",
                leave=True,
                console=self.console,
            ):
                output, item_cost = future.result()
                total_cost += item_cost
                if output is not None:
                    results.append(output)

        if self.config.get("persist_intermediates", False):
            for result in results:
                key = tuple(result[k] for k in self.config["reduce_key"])
                if key in self.intermediates:
                    result[f"_{self.config['name']}_intermediates"] = (
```

```
                        self.intermediates[key]
                    )

        if self.status:
            self.status.start()

        return results, total_cost

    def _cluster_based_sampling(
        self, group_list: list[dict], value_sampling: dict, sample_size:
int
    ) -> tuple[list[dict], float]:
        if sample_size >= len(group_list):
            return group_list, 0

        clusters, cost = cluster_documents(
            group_list, value_sampling, sample_size, self.runner.api
        )

        sampled_items = []
        idx_added_already = set()
        num_clusters = len(clusters)
        for i in range(sample_size):
            # Add a random item from the cluster
            idx = i % num_clusters

            # Skip if there are no items in the cluster
            if len(clusters[idx]) == 0:
                continue

            if len(clusters[idx]) == 1:
                # If there's only one item in the cluster, add it
directly if we haven't already
                if idx not in idx_added_already:
                    sampled_items.append(clusters[idx][0])
                continue

            random_choice_idx = random.randint(0, len(clusters[idx]) - 1)
            max_attempts = 10
            while random_choice_idx in idx_added_already and max_attempts
> 0:
                random_choice_idx = random.randint(0, len(clusters[idx])
- 1)
                max_attempts -= 1
            idx_added_already.add(random_choice_idx)
            sampled_items.append(clusters[idx][random_choice_idx])

        return sampled_items, cost

    def _semantic_similarity_sampling(
        self, key: tuple, group_list: list[dict], value_sampling: dict,
sample_size: int
    ) -> tuple[list[dict], float]:
        embedding_model = value_sampling["embedding_model"]
        query_text = strict_render(
            value_sampling["query_text"],
            {"reduce_key": dict(zip(self.config["reduce_key"], key))},
        )

        embeddings, cost = get_embeddings_for_clustering(
            group_list, value_sampling, self.runner.api
```

```
        )

        query_response = self.runner.api.gen_embedding(embedding_model,
[query_text])
        query_embedding = query_response["data"][0]["embedding"]
        cost += completion_cost(query_response)

        from sklearn.metrics.pairwise import cosine_similarity

        similarities = cosine_similarity([query_embedding], embeddings)
[0]

        top_k_indices = np.argsort(similarities)[-sample_size:]

        return [group_list[i] for i in top_k_indices], cost

    def _parallel_fold_and_merge(
        self, key: tuple, group_list: list[dict]
    ) -> tuple[dict | None, float]:
        """
        Perform parallel folding and merging on a group of items.

        This method implements a strategy that combines parallel folding
of input items
        and merging of intermediate results to efficiently process large
groups. It works as follows:
        1. The input group is initially divided into smaller batches for
efficient processing.
        2. The method performs an initial round of folding operations on
these batches.
        3. After the first round of folds, a few merges are performed to
estimate the merge runtime.
        4. Based on the estimated merge runtime and observed fold
runtime, it calculates the optimal number of parallel folds. Subsequent
rounds of folding are then performed concurrently, with the number of
parallel folds determined by the runtime estimates.
        5. The folding process repeats in rounds, progressively reducing
the number of items to be processed.
        6. Once all folding operations are complete, the method
recursively performs final merges on the fold results to combine them
into a final result.
        7. Throughout this process, the method may adjust the number of
parallel folds based on updated performance metrics (i.e., fold and merge
runtimes) to maintain efficiency.

        Args:
            key (tuple): The reduce key tuple for the group.
            group_list (list[dict]): The list of items in the group to be
processed.

        Returns:
            tuple[dict | None, float]: A tuple containing the final
merged result (or None if processing failed)
            and the total cost of the operation.
        """
        fold_batch_size = self.config["fold_batch_size"]
        merge_batch_size = self.config["merge_batch_size"]
        total_cost = 0
        prompts = []

        def calculate_num_parallel_folds():
```

```
            fold_time, fold_default = self.get_fold_time()
            merge_time, merge_default = self.get_merge_time()
            num_group_items = len(group_list)
            return (
                max(
                    1,
                    int(
                        (fold_time * num_group_items *
math.log(merge_batch_size))
                        / (fold_batch_size * merge_time)
                    ),
                ),
                fold_default or merge_default,
            )

        num_parallel_folds, used_default_times =
calculate_num_parallel_folds()
        fold_results = []
        remaining_items = group_list

        if self.config.get("persist_intermediates", False):
            self.intermediates[key] = []
            iter_count = 0

        # Parallel folding and merging
        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            while remaining_items:
                # Folding phase
                fold_futures = []
                for i in range(min(num_parallel_folds,
len(remaining_items))):
                    batch = remaining_items[:fold_batch_size]
                    remaining_items = remaining_items[fold_batch_size:]
                    current_output = fold_results[i] if i <
len(fold_results) else None
                    fold_futures.append(
                        executor.submit(
                            self._increment_fold, key, batch,
current_output
                        )
                    )

                new_fold_results = []
                for future in as_completed(fold_futures):
                    result, prompt, cost = future.result()
                    total_cost += cost
                    prompts.append(prompt)
                    if result is not None:
                        new_fold_results.append(result)
                        if self.config.get("persist_intermediates",
False):
                            self.intermediates[key].append(
                                {
                                    "iter": iter_count,
                                    "intermediate": result,
                                    "scratchpad":
result["updated_scratchpad"],
                                }
                            )
                            iter_count += 1
```

```
                # Update fold_results with new results
                fold_results = new_fold_results +
fold_results[len(new_fold_results) :]

                # Single pass merging phase
                if (
                    len(self.merge_times) < self.min_samples
                    and len(fold_results) >= merge_batch_size
                ):
                    merge_futures = []
                    for i in range(0, len(fold_results),
merge_batch_size):
                        batch = fold_results[i : i + merge_batch_size]
                        merge_futures.append(
                            executor.submit(self._merge_results, key,
batch)
                        )

                    new_results = []
                    for future in as_completed(merge_futures):
                        result, prompt, cost = future.result()
                        total_cost += cost
                        prompts.append(prompt)
                        if result is not None:
                            new_results.append(result)
                            if self.config.get("persist_intermediates",
False):
                                self.intermediates[key].append(
                                    {
                                        "iter": iter_count,
                                        "intermediate": result,
                                        "scratchpad": None,
                                    }
                                )
                                iter_count += 1

                    fold_results = new_results

                # Recalculate num_parallel_folds if we used default times
                if used_default_times:
                    new_num_parallel_folds, used_default_times = (
                        calculate_num_parallel_folds()
                    )
                    if not used_default_times:
                        self.console.log(
                            f"Recalculated num_parallel_folds from
{num_parallel_folds} to {new_num_parallel_folds}"
                        )
                        num_parallel_folds = new_num_parallel_folds

        # Final merging if needed
        while len(fold_results) > 1:
            self.console.log(f"Finished folding! Merging
{len(fold_results)} items.")
            with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
                merge_futures = []
                for i in range(0, len(fold_results), merge_batch_size):
                    batch = fold_results[i : i + merge_batch_size]
                    merge_futures.append(
```

```
                        executor.submit(self._merge_results, key, batch)
                    )

                new_results = []
                for future in as_completed(merge_futures):
                    result, prompt, cost = future.result()
                    total_cost += cost
                    prompts.append(prompt)
                    if result is not None:
                        new_results.append(result)
                        if self.config.get("persist_intermediates",
False):
                            self.intermediates[key].append(
                                {
                                    "iter": iter_count,
                                    "intermediate": result,
                                    "scratchpad": None,
                                }
                            )
                            iter_count += 1

                fold_results = new_results

        return (
            (fold_results[0], prompts, total_cost)
            if fold_results
            else (None, prompts, total_cost)
        )

    def _incremental_reduce(
        self, key: tuple, group_list: list[dict]
    ) -> tuple[dict | None, list[str], float]:
        """
        Perform an incremental reduce operation on a group of items.

        This method processes the group in batches, incrementally folding
the results.

        Args:
            key (tuple): The reduce key tuple for the group.
            group_list (list[dict]): The list of items in the group to be
processed.

        Returns:
            tuple[dict | None, list[str], float]: A tuple containing the
final reduced result (or None if processing failed),
            the list of prompts used, and the total cost of the
operation.
        """
        fold_batch_size = self.config["fold_batch_size"]
        total_cost = 0
        current_output = None
        prompts = []

        # Calculate and log the number of folds to be performed
        num_folds = (len(group_list) + fold_batch_size - 1) //
fold_batch_size

        scratchpad = ""
        if self.config.get("persist_intermediates", False):
            self.intermediates[key] = []
```

```
            iter_count = 0

        for i in range(0, len(group_list), fold_batch_size):
            # Log the current iteration and total number of folds
            current_fold = i // fold_batch_size + 1
            if self.config.get("verbose", False):
                self.console.log(
                    f"Processing fold {current_fold} of {num_folds} for
group with key {key}"
                )
            batch = group_list[i : i + fold_batch_size]

            folded_output, prompt, fold_cost = self._increment_fold(
                key, batch, current_output, scratchpad
            )
            total_cost += fold_cost
            prompts.append(prompt)

            if folded_output is None:
                continue

            if self.config.get("persist_intermediates", False):
                self.intermediates[key].append(
                    {
                        "iter": iter_count,
                        "intermediate": folded_output,
                        "scratchpad":
folded_output.get("updated_scratchpad", ""),
                    }
                )
                iter_count += 1

            # Pop off updated_scratchpad
            if "updated_scratchpad" in folded_output:
                scratchpad = folded_output["updated_scratchpad"]
                if self.config.get("verbose", False):
                    self.console.log(
                        f"Updated scratchpad for fold {current_fold}:
{scratchpad}"
                    )
                del folded_output["updated_scratchpad"]

            current_output = folded_output

        return current_output, prompts, total_cost

    def validation_fn(self, response: dict[str, Any]):
        structured_mode = (
            self.config.get("output", {}).get("mode")
            == OutputMode.STRUCTURED_OUTPUT.value
        )
        output = self.runner.api.parse_llm_response(
            response,
            schema=self.config["output"]["schema"],
            use_structured_output=structured_mode,
        )[0]
        if self.runner.api.validate_output(self.config, output,
self.console):
            return output, True
        return output, False

```

```python
    def _increment_fold(
        self,
        key: tuple,
        batch: list[dict],
        current_output: dict | None,
        scratchpad: str | None = None,
    ) -> tuple[dict | None, str, float]:
        """
        Perform an incremental fold operation on a batch of items.

        This method folds a batch of items into the current output using the fold prompt.

        Args:
            key (tuple): The reduce key tuple for the group.
            batch (list[dict]): The batch of items to be folded.
            current_output (dict | None): The current accumulated output, if any.
            scratchpad (str | None): The scratchpad to use for the fold operation.
        Returns:
            tuple[dict | None, str, float]: A tuple containing the folded output (or None if processing failed),
            the prompt used, and the cost of the fold operation.
        """
        if current_output is None:
            return self._batch_reduce(key, batch, scratchpad)

        start_time = time.time()
        fold_prompt = strict_render(
            self.config["fold_prompt"],
            {
                "inputs": batch,
                "output": current_output,
                "reduce_key": dict(zip(self.config["reduce_key"], key)),
            },
        )

        response = self.runner.api.call_llm(
            self.config.get("model", self.default_model),
            "reduce",
            [{"role": "user", "content": fold_prompt}],
            self.config["output"]["schema"],
            scratchpad=scratchpad,
            timeout_seconds=self.config.get("timeout", 120),

max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
            validation_config=(
                {
                    "num_retries": self.num_retries_on_validate_failure,
                    "val_rule": self.config.get("validate", []),
                    "validation_fn": self.validation_fn,
                }
                if self.config.get("validate", None)
                else None
            ),
            bypass_cache=self.config.get("bypass_cache", self.bypass_cache),
            verbose=self.config.get("verbose", False),

litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
```

```
{}),
            op_config=self.config,
        )

        end_time = time.time()
        self._update_fold_time(end_time - start_time)

        if response.validated:
            structured_mode = (
                self.config.get("output", {}).get("mode")
                == OutputMode.STRUCTURED_OUTPUT.value
            )
            folded_output = self.runner.api.parse_llm_response(
                response.response,
                schema=self.config["output"]["schema"],
                manually_fix_errors=self.manually_fix_errors,
                use_structured_output=structured_mode,
            )[0]

            folded_output.update(dict(zip(self.config["reduce_key"],
key)))
            fold_cost = response.total_cost

            return folded_output, fold_prompt, fold_cost

        return None, fold_prompt, fold_cost

    def _merge_results(
        self, key: tuple, outputs: list[dict]
    ) -> tuple[dict | None, str, float]:
        """
        Merge multiple outputs into a single result.

        This method merges a list of outputs using the merge prompt.

        Args:
            key (tuple): The reduce key tuple for the group.
            outputs (list[dict]): The list of outputs to be merged.

        Returns:
            tuple[dict | None, str, float]: A tuple containing the merged
output (or None if processing failed),
            the prompt used, and the cost of the merge operation.
        """
        start_time = time.time()
        merge_prompt = strict_render(
            self.config["merge_prompt"],
            {
                "outputs": outputs,
                "reduce_key": dict(zip(self.config["reduce_key"], key)),
            },
        )
        response = self.runner.api.call_llm(
            self.config.get("model", self.default_model),
            "merge",
            [{"role": "user", "content": merge_prompt}],
            self.config["output"]["schema"],
            timeout_seconds=self.config.get("timeout", 120),

max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
            validation_config=(
```

```
                {
                    "num_retries": self.num_retries_on_validate_failure,
                    "val_rule": self.config.get("validate", []),
                    "validation_fn": self.validation_fn,
                }
                if self.config.get("validate", None)
                else None
            ),
            bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),
            verbose=self.config.get("verbose", False),

litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
{}),
            op_config=self.config,
        )

        end_time = time.time()
        self._update_merge_time(end_time - start_time)

        if response.validated:
            structured_mode = (
                self.config.get("output", {}).get("mode")
                == OutputMode.STRUCTURED_OUTPUT.value
            )
            merged_output = self.runner.api.parse_llm_response(
                response.response,
                schema=self.config["output"]["schema"],
                manually_fix_errors=self.manually_fix_errors,
                use_structured_output=structured_mode,
            )[0]
            merged_output.update(dict(zip(self.config["reduce_key"],
key)))
            merge_cost = response.total_cost
            return merged_output, merge_prompt, merge_cost

        return None, merge_prompt, merge_cost

    def get_fold_time(self) -> tuple[float, bool]:
        """
        Get the average fold time or a default value.

        Returns:
            tuple[float, bool]: A tuple containing the average fold time
(or default) and a boolean
            indicating whether the default value was used.
        """
        if "fold_time" in self.config:
            return self.config["fold_time"], False
        with self.lock:
            if len(self.fold_times) >= self.min_samples:
                return sum(self.fold_times) / len(self.fold_times), False
        return 1.0, True  # Default to 1 second if no data is available

    def get_merge_time(self) -> tuple[float, bool]:
        """
        Get the average merge time or a default value.

        Returns:
            tuple[float, bool]: A tuple containing the average merge time
(or default) and a boolean
```

```
                indicating whether the default value was used.
            """
            if "merge_time" in self.config:
                return self.config["merge_time"], False
            with self.lock:
                if len(self.merge_times) >= self.min_samples:
                    return sum(self.merge_times) / len(self.merge_times),
    False
            return 1.0, True  # Default to 1 second if no data is available

        def _update_fold_time(self, time: float) -> None:
            """
            Update the fold time statistics.

            Args:
                time (float): The time taken for a fold operation.
            """
            with self.lock:
                self.fold_times.append(time)

        def _update_merge_time(self, time: float) -> None:
            """
            Update the merge time statistics.

            Args:
                time (float): The time taken for a merge operation.
            """
            with self.lock:
                self.merge_times.append(time)

        def _batch_reduce(
            self, key: tuple, group_list: list[dict], scratchpad: str | None
    = None
        ) -> tuple[dict | None, str, float]:
            """
            Perform a batch reduce operation on a group of items.

            This method reduces a group of items into a single output using
    the reduce prompt.

            Args:
                key (tuple): The reduce key tuple for the group.
                group_list (list[dict]): The list of items to be reduced.
                scratchpad (str | None): The scratchpad to use for the reduce
    operation.
            Returns:
                tuple[dict | None, str, float]: A tuple containing the
    reduced output (or None if processing failed),
                the prompt used, and the cost of the reduce operation.
            """
            prompt = strict_render(
                self.config["prompt"],
                {
                    "reduce_key": dict(zip(self.config["reduce_key"], key)),
                    "inputs": group_list,
                },
            )
            item_cost = 0

            response = self.runner.api.call_llm(
                self.config.get("model", self.default_model),
```

```
                "reduce",
                [{"role": "user", "content": prompt}],
                self.config["output"]["schema"],
                scratchpad=scratchpad,
                timeout_seconds=self.config.get("timeout", 120),

    max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
                bypass_cache=self.config.get("bypass_cache",
    self.bypass_cache),
                validation_config=(
                    {
                        "num_retries": self.num_retries_on_validate_failure,
                        "val_rule": self.config.get("validate", []),
                        "validation_fn": self.validation_fn,
                    }
                    if self.config.get("validate", None)
                    else None
                ),
                gleaning_config=self.config.get("gleaning", None),
                verbose=self.config.get("verbose", False),

    litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
    {}),
                op_config=self.config,
            )

            item_cost += response.total_cost

            if response.validated:
                structured_mode = (
                    self.config.get("output", {}).get("mode")
                    == OutputMode.STRUCTURED_OUTPUT.value
                )
                output = self.runner.api.parse_llm_response(
                    response.response,
                    schema=self.config["output"]["schema"],
                    manually_fix_errors=self.manually_fix_errors,
                    use_structured_output=structured_mode,
                )[0]
                output.update(dict(zip(self.config["reduce_key"], key)))

                return output, prompt, item_cost
            return None, prompt, item_cost
```

`__init__(*args, **kwargs)`

Initialize the ReduceOperation.

Parameters:

Name	Type	Description	Default
`*args`		Variable length argument list.	`()`
`**kwargs`		Arbitrary keyword arguments.	`{}`

Source code in `docetl/operations/reduce.py`

```python
def __init__(self, *args, **kwargs):
    """
    Initialize the ReduceOperation.

    Args:
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.
    """
    super().__init__(*args, **kwargs)
    self.min_samples = 5
    self.max_samples = 1000
    self.fold_times = deque(maxlen=self.max_samples)
    self.merge_times = deque(maxlen=self.max_samples)
    self.lock = Lock()
    self.config["reduce_key"] = (
        [self.config["reduce_key"]]
        if isinstance(self.config["reduce_key"], str)
        else self.config["reduce_key"]
    )
    self.intermediates = {}
    self.lineage_keys = self.config.get("output", {}).get("lineage", [])
```

`execute(input_data)`

Execute the reduce operation on the provided input data.

This method sorts and groups the input data by the reduce key(s), then processes each group using either parallel fold and merge, incremental reduce, or batch reduce strategies.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	The input data to process.	*required*

Returns:

Type	Description
`tuple[list[dict], float]`	tuple[list[dict], float]: A tuple containing the processed results and the total cost of the operation.

Source code in `docetl/operations/reduce.py`

```python
def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
    """
    Execute the reduce operation on the provided input data.

    This method sorts and groups the input data by the reduce key(s), then processes each group
    using either parallel fold and merge, incremental reduce, or batch reduce strategies.

    Args:
        input_data (list[dict]): The input data to process.

    Returns:
        tuple[list[dict], float]: A tuple containing the processed results and the total cost of the operation.
    """
    if self.config.get("gleaning", {}).get("validation_prompt", None):
        self.console.log(
            f"Using gleaning with validation prompt: {self.config.get('gleaning', {}).get('validation_prompt', '')}"
        )

    reduce_keys = self.config["reduce_key"]
    if isinstance(reduce_keys, str):
        reduce_keys = [reduce_keys]
    input_schema = self.config.get("input", {}).get("schema", {})

    if self.status:
        self.status.stop()

    # Check if we need to group everything into one group
    if reduce_keys == ["_all"] or reduce_keys == "_all":
        grouped_data = [("_all", input_data)]
    else:
        # Group the input data by the reduce key(s) while maintaining original order
        def get_group_key(item):
            key_values = []
            for key in reduce_keys:
                value = item[key]
                # Special handling for list-type values
                if isinstance(value, list):
                    key_values.append(
                        tuple(sorted(value))
                    )  # Convert list to sorted tuple
                else:
                    key_values.append(value)
            return tuple(key_values)

        grouped_data = {}
        for item in input_data:
            key = get_group_key(item)
            if key not in grouped_data:
                grouped_data[key] = []
            grouped_data[key].append(item)

        # Convert the grouped data to a list of tuples
```

```
        grouped_data = list(grouped_data.items())

    def process_group(
        key: tuple, group_elems: list[dict]
    ) -> tuple[dict | None, float]:
        if input_schema:
            group_list = [
                {k: item[k] for k in input_schema.keys() if k in item}
                for item in group_elems
            ]
        else:
            group_list = group_elems

        total_cost = 0.0

        # Apply value sampling if enabled
        value_sampling = self.config.get("value_sampling", {})
        if value_sampling.get("enabled", False):
            sample_size = min(value_sampling["sample_size"],
len(group_list))
            method = value_sampling["method"]

            if method == "random":
                group_sample = random.sample(group_list, sample_size)
                group_sample.sort(key=lambda x: group_list.index(x))
            elif method == "first_n":
                group_sample = group_list[:sample_size]
            elif method == "cluster":
                group_sample, embedding_cost =
self._cluster_based_sampling(
                    group_list, value_sampling, sample_size
                )
                group_sample.sort(key=lambda x: group_list.index(x))
                total_cost += embedding_cost
            elif method == "sem_sim":
                group_sample, embedding_cost =
self._semantic_similarity_sampling(
                    key, group_list, value_sampling, sample_size
                )
                group_sample.sort(key=lambda x: group_list.index(x))
                total_cost += embedding_cost

            group_list = group_sample

        # Only execute merge-based plans if associative = True
        if "merge_prompt" in self.config and
self.config.get("associative", True):
            result, prompts, cost = self._parallel_fold_and_merge(key,
group_list)
        elif self.config.get("fold_batch_size", None) and
self.config.get(
            "fold_batch_size"
        ) >= len(group_list):
            # If the fold batch size is greater than or equal to the
number of items in the group,
            # we can just run a single fold operation
            result, prompt, cost = self._batch_reduce(key, group_list)
            prompts = [prompt]
        elif "fold_prompt" in self.config:
            result, prompts, cost = self._incremental_reduce(key,
group_list)
```

```
        else:
            result, prompt, cost = self._batch_reduce(key, group_list)
            prompts = [prompt]

        total_cost += cost

        # Add the counts of items in the group to the result
        result[f"_counts_prereduce_{self.config['name']}"] =
len(group_elems)

        if self.config.get("enable_observability", False):
            # Add the _observability_{self.config['name']} key to the
result
            result[f"_observability_{self.config['name']}"] = {"prompts":
prompts}

        # Apply pass-through at the group level
        if (
            result is not None
            and self.config.get("pass_through", False)
            and group_elems
        ):
            for k, v in group_elems[0].items():
                if k not in self.config["output"]["schema"] and k not in
result:
                    result[k] = v

        # Add lineage information
        if result is not None and self.lineage_keys:
            lineage = []
            for item in group_elems:
                lineage_item = {
                    k: item.get(k) for k in self.lineage_keys if k in
item
                }
                if lineage_item:
                    lineage.append(lineage_item)
            result[f"{self.config['name']}_lineage"] = lineage

        return result, total_cost

    with ThreadPoolExecutor(max_workers=self.max_threads) as executor:
        futures = [
            executor.submit(process_group, key, group)
            for key, group in grouped_data
        ]
        results = []
        total_cost = 0
        for future in rich_as_completed(
            futures,
            total=len(futures),
            desc=f"Processing {self.config['name']} (reduce) on all
documents",
            leave=True,
            console=self.console,
        ):
            output, item_cost = future.result()
            total_cost += item_cost
            if output is not None:
                results.append(output)
```

```
        if self.config.get("persist_intermediates", False):
            for result in results:
                key = tuple(result[k] for k in self.config["reduce_key"])
                if key in self.intermediates:
                    result[f"_{self.config['name']}_intermediates"] = (
                        self.intermediates[key]
                    )

        if self.status:
            self.status.start()

        return results, total_cost
```

`get_fold_time()`

Get the average fold time or a default value.

Returns:

Type	Description
`float`	tuple[float, bool]: A tuple containing the average fold time (or default) and a boolean
`bool`	indicating whether the default value was used.

Source code in `docetl/operations/reduce.py`

```
810 def get_fold_time(self) -> tuple[float, bool]:
811     """
812     Get the average fold time or a default value.
813
814     Returns:
815         tuple[float, bool]: A tuple containing the average fold time (or
816 default) and a boolean
817         indicating whether the default value was used.
818     """
819     if "fold_time" in self.config:
820         return self.config["fold_time"], False
821     with self.lock:
822         if len(self.fold_times) >= self.min_samples:
823             return sum(self.fold_times) / len(self.fold_times), False
        return 1.0, True  # Default to 1 second if no data is available
```

`get_merge_time()`

Get the average merge time or a default value.

Returns:

Type	Description
`float`	tuple[float, bool]: A tuple containing the average merge time (or default) and a boolean
`bool`	indicating whether the default value was used.

Source code in `docetl/operations/reduce.py`

```
825  def get_merge_time(self) -> tuple[float, bool]:
826      """
827      Get the average merge time or a default value.
828
829      Returns:
830          tuple[float, bool]: A tuple containing the average merge time (or
831  default) and a boolean
832          indicating whether the default value was used.
833      """
834      if "merge_time" in self.config:
835          return self.config["merge_time"], False
836      with self.lock:
837          if len(self.merge_times) >= self.min_samples:
838              return sum(self.merge_times) / len(self.merge_times), False
         return 1.0, True  # Default to 1 second if no data is available
```

`docetl.operations.map.ParallelMapOperation`

Bases: `BaseOperation`

Source code in `docetl/operations/map.py`

```
class ParallelMapOperation(BaseOperation):
    class schema(BaseOperation.schema):
        type: str = "parallel_map"
        prompts: list[dict[str, Any]] | None = None
        output: dict[str, Any] | None = None
        drop_keys: list[str] | None = None
        enable_observability: bool = False
        pdf_url_key: str | None = None

        @field_validator("prompts")
        def validate_prompts(cls, v):
            if v is not None:
                if not v:
                    raise ValueError("The 'prompts' list cannot be
empty")

                for i, prompt_config in enumerate(v):
                    # Validate required keys exist
                    if "prompt" not in prompt_config:
                        raise ValueError(
                            f"Missing required key 'prompt' in prompt
configuration {i}"
                        )
                    if "output_keys" not in prompt_config:
                        raise ValueError(
                            f"Missing required key 'output_keys' in
prompt configuration {i}"
                        )

                    # Validate output_keys is not empty
                    if not prompt_config["output_keys"]:
                        raise ValueError(
                            f"'output_keys' list in prompt configuration
{i} cannot be empty"
                        )

                    # Check if the prompt is a valid Jinja2 template
                    try:
                        Template(prompt_config["prompt"])
                    except Exception as e:
                        raise ValueError(
                            f"Invalid Jinja2 template in prompt
configuration {i}: {str(e)}"
                        ) from e
            return v

        @model_validator(mode="after")
        def validate_prompt_requirements(self):
            # If drop_keys is not specified, prompts must be present
            if not self.drop_keys and not self.prompts:
                raise ValueError(
                    "If 'drop_keys' is not specified, 'prompts' must be
present in the configuration"
                )

            # Check if all output schema keys are covered by the prompts
            if self.prompts and self.output and "schema" in self.output:
```

```
                output_schema = self.output["schema"]
                output_keys_covered = set()
                for prompt_config in self.prompts:

output_keys_covered.update(prompt_config["output_keys"])

                missing_keys = set(output_schema.keys()) -
output_keys_covered
                if missing_keys:
                    raise ValueError(
                        f"The following output schema keys are not
covered by any prompt: {missing_keys}"
                    )

            return self

    def __init__(
        self,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)

    def execute(self, input_data: list[dict]) -> tuple[list[dict],
float]:
        """
        Executes the parallel map operation on the provided input data.

        Args:
            input_data (list[dict]): The input data to process.

        Returns:
            tuple[list[dict], float]: A tuple containing the processed
results and the total cost of the operation.

        This method performs the following steps:
        1. If prompts are specified, it processes each input item using
multiple prompts in parallel
        2. Aggregates results from different prompts for each input item
        3. Validates the combined output for each item
        4. If drop_keys is specified, it drops the specified keys from
each document
        5. Calculates total cost of the operation
        """
        results = {}
        total_cost = 0
        output_schema = self.config.get("output", {}).get("schema", {})

        # Check if there's no prompt and only drop_keys
        if "prompts" not in self.config and "drop_keys" in self.config:
            # If only drop_keys is specified, simply drop the keys and
return
            dropped_results = []
            for item in input_data:
                new_item = {
                    k: v for k, v in item.items() if k not in
self.config["drop_keys"]
                }
                dropped_results.append(new_item)
            return dropped_results, 0.0  # Return the modified data with
no cost
```

```
640
641         if self.status:
642             self.status.stop()
643
644         def process_prompt(item, prompt_config):
645             prompt = strict_render(prompt_config["prompt"], {"input":
646 item})
647             messages = [{"role": "user", "content": prompt}]
648             if self.config.get("pdf_url_key", None):
649                 try:
650                     pdf_url = item[self.config["pdf_url_key"]]
651                 except KeyError:
652                     raise ValueError(
653                         f"PDF URL key '{self.config['pdf_url_key']}' not
654 found in input data"
655                     )
656                 # Download content
657                 if pdf_url.startswith("http"):
658                     file_data = requests.get(pdf_url).content
659                 else:
660                     with open(pdf_url, "rb") as f:
661                         file_data = f.read()
662                 encoded_file = base64.b64encode(file_data).decode("utf-
663 8")
664                 base64_url = f"data:application/pdf;base64,
665 {encoded_file}"
666
667                 messages[0]["content"] = [
668                     {"type": "image_url", "image_url": {"url":
669 base64_url}},
670                     {"type": "text", "text": prompt},
671                 ]
672
673             local_output_schema = {
674                 key: output_schema.get(key, "string")
675                 for key in prompt_config["output_keys"]
676             }
677             model = prompt_config.get("model", self.default_model)
678             if not model:
679                 model = self.default_model
680
681             # Start of Selection
682             # If there are tools, we need to pass in the tools
683             response = self.runner.api.call_llm(
684                 model,
685                 "parallel_map",
686                 messages,
687                 local_output_schema,
688                 tools=prompt_config.get("tools", None),
689                 timeout_seconds=self.config.get("timeout", 120),
690
691 max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
692                 gleaning_config=prompt_config.get("gleaning", None),
693                 bypass_cache=self.config.get("bypass_cache",
694 self.bypass_cache),
695                 litellm_completion_kwargs=self.config.get(
696                     "litellm_completion_kwargs", {}
697                 ),
698                 op_config=self.config,
699             )
700             structured_mode = (
```

```
                self.config.get("output", {}).get("mode")
                == OutputMode.STRUCTURED_OUTPUT.value
            )
            output = self.runner.api.parse_llm_response(
                response.response,
                schema=local_output_schema,
                tools=prompt_config.get("tools", None),
                manually_fix_errors=self.manually_fix_errors,
                use_structured_output=structured_mode,
            )[0]
            return output, prompt, response.total_cost

        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            if "prompts" in self.config:
                # Create all futures at once
                all_futures = [
                    executor.submit(process_prompt, item, prompt_config)
                    for item in input_data
                    for prompt_config in self.config["prompts"]
                ]

                # Process results in order
                for i in tqdm(
                    range(len(all_futures)),
                    desc="Processing parallel map items",
                ):
                    future = all_futures[i]
                    output, prompt, cost = future.result()
                    total_cost += cost

                    # Determine which item this future corresponds to
                    item_index = i // len(self.config["prompts"])
                    prompt_index = i % len(self.config["prompts"])

                    # Initialize or update the item_result
                    if prompt_index == 0:
                        item_result = input_data[item_index].copy()
                        results[item_index] = item_result

                    # Fetch the item_result
                    item_result = results[item_index]

                    if self.config.get("enable_observability", False):
                        if f"_observability_{self.config['name']}" not in
    item_result:

    item_result[f"_observability_{self.config['name']}"] = {}

    item_result[f"_observability_{self.config['name']}"].update(
                                {f"prompt_{prompt_index}": prompt}
                            )

                    # Update the item_result with the output
                    item_result.update(output)

            else:
                results = {i: item.copy() for i, item in
    enumerate(input_data)}

        # Apply drop_keys if specified
```

```
            if "drop_keys" in self.config:
                drop_keys = self.config["drop_keys"]
                for item in results.values():
                    for key in drop_keys:
                        item.pop(key, None)

            if self.status:
                self.status.start()

            # Return the results in order
            return [results[i] for i in range(len(input_data)) if i in
    results], total_cost
```

`execute(input_data)`

Executes the parallel map operation on the provided input data.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	The input data to process.	*required*

Returns:

Type	Description
`tuple[list[dict], float]`	tuple[list[dict], float]: A tuple containing the processed results and the total cost of the operation.

This method performs the following steps: 1. If prompts are specified, it processes each input item using multiple prompts in parallel 2. Aggregates results from different prompts for each input item 3. Validates the combined output for each item 4. If drop_keys is specified, it drops the specified keys from each document 5. Calculates total cost of the operation

Source code in `docetl/operations/map.py`

```
def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
    """
    Executes the parallel map operation on the provided input data.

    Args:
        input_data (list[dict]): The input data to process.

    Returns:
        tuple[list[dict], float]: A tuple containing the processed
results and the total cost of the operation.

    This method performs the following steps:
    1. If prompts are specified, it processes each input item using
multiple prompts in parallel
    2. Aggregates results from different prompts for each input item
    3. Validates the combined output for each item
    4. If drop_keys is specified, it drops the specified keys from each
document
    5. Calculates total cost of the operation
    """
    results = {}
    total_cost = 0
    output_schema = self.config.get("output", {}).get("schema", {})

    # Check if there's no prompt and only drop_keys
    if "prompts" not in self.config and "drop_keys" in self.config:
        # If only drop_keys is specified, simply drop the keys and return
        dropped_results = []
        for item in input_data:
            new_item = {
                k: v for k, v in item.items() if k not in
self.config["drop_keys"]
            }
            dropped_results.append(new_item)
        return dropped_results, 0.0  # Return the modified data with no
cost

    if self.status:
        self.status.stop()

    def process_prompt(item, prompt_config):
        prompt = strict_render(prompt_config["prompt"], {"input": item})
        messages = [{"role": "user", "content": prompt}]
        if self.config.get("pdf_url_key", None):
            try:
                pdf_url = item[self.config["pdf_url_key"]]
            except KeyError:
                raise ValueError(
                    f"PDF URL key '{self.config['pdf_url_key']}' not
found in input data"
                )
            # Download content
            if pdf_url.startswith("http"):
                file_data = requests.get(pdf_url).content
            else:
                with open(pdf_url, "rb") as f:
                    file_data = f.read()
```

```
650             encoded_file = base64.b64encode(file_data).decode("utf-8")
651             base64_url = f"data:application/pdf;base64,{encoded_file}"
652 
653             messages[0]["content"] = [
654                 {"type": "image_url", "image_url": {"url": base64_url}},
655                 {"type": "text", "text": prompt},
656             ]
657 
658         local_output_schema = {
659             key: output_schema.get(key, "string")
660             for key in prompt_config["output_keys"]
661         }
662         model = prompt_config.get("model", self.default_model)
663         if not model:
664             model = self.default_model
665 
666         # Start of Selection
667         # If there are tools, we need to pass in the tools
668         response = self.runner.api.call_llm(
669             model,
670             "parallel_map",
671             messages,
672             local_output_schema,
673             tools=prompt_config.get("tools", None),
674             timeout_seconds=self.config.get("timeout", 120),
675 
676 max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
677             gleaning_config=prompt_config.get("gleaning", None),
678             bypass_cache=self.config.get("bypass_cache",
679 self.bypass_cache),
680             litellm_completion_kwargs=self.config.get(
681                 "litellm_completion_kwargs", {}
682             ),
683             op_config=self.config,
684         )
685         structured_mode = (
686             self.config.get("output", {}).get("mode")
687             == OutputMode.STRUCTURED_OUTPUT.value
688         )
689         output = self.runner.api.parse_llm_response(
690             response.response,
691             schema=local_output_schema,
692             tools=prompt_config.get("tools", None),
693             manually_fix_errors=self.manually_fix_errors,
694             use_structured_output=structured_mode,
695         )[0]
696         return output, prompt, response.total_cost
697 
698     with ThreadPoolExecutor(max_workers=self.max_threads) as executor:
699         if "prompts" in self.config:
700             # Create all futures at once
701             all_futures = [
702                 executor.submit(process_prompt, item, prompt_config)
703                 for item in input_data
704                 for prompt_config in self.config["prompts"]
705             ]
706 
707             # Process results in order
708             for i in tqdm(
709                 range(len(all_futures)),
710                 desc="Processing parallel map items",
```

```
            ):
                future = all_futures[i]
                output, prompt, cost = future.result()
                total_cost += cost

                # Determine which item this future corresponds to
                item_index = i // len(self.config["prompts"])
                prompt_index = i % len(self.config["prompts"])

                # Initialize or update the item_result
                if prompt_index == 0:
                    item_result = input_data[item_index].copy()
                    results[item_index] = item_result

                # Fetch the item_result
                item_result = results[item_index]

                if self.config.get("enable_observability", False):
                    if f"_observability_{self.config['name']}" not in
item_result:

item_result[f"_observability_{self.config['name']}"] = {}

item_result[f"_observability_{self.config['name']}"].update(
                        {f"prompt_{prompt_index}": prompt}
                    )

                # Update the item_result with the output
                item_result.update(output)

        else:
            results = {i: item.copy() for i, item in
enumerate(input_data)}

    # Apply drop_keys if specified
    if "drop_keys" in self.config:
        drop_keys = self.config["drop_keys"]
        for item in results.values():
            for key in drop_keys:
                item.pop(key, None)

    if self.status:
        self.status.start()

    # Return the results in order
    return [results[i] for i in range(len(input_data)) if i in results],
total_cost
```

`docetl.operations.filter.FilterOperation`

Bases: `MapOperation`

Source code in `docetl/operations/filter.py`

```python
class FilterOperation(MapOperation):
    class schema(MapOperation.schema):
        type: str = "filter"
        prompt: str
        output: dict[str, Any]

        @model_validator(mode="after")
        def validate_filter_output_schema(self):
            # Check that schema exists and has the right structure for filtering
            schema_dict = self.output["schema"]

            # Filter out _short_explanation for validation
            schema = {k: v for k, v in schema_dict.items() if k != "_short_explanation"}
            if len(schema) != 1:
                raise ValueError(
                    "The 'schema' in 'output' configuration must have exactly one key-value pair that maps to a boolean value"
                )

            key, value = next(iter(schema.items()))
            if value not in ["bool", "boolean"]:
                raise TypeError(
                    f"The value in the 'schema' must be of type bool, got {value}"
                )

            return self

    def execute(
        self, input_data: list[dict], is_build: bool = False
    ) -> tuple[list[dict], float]:
        """
        Executes the filter operation on the input data.

        Args:
            input_data (list[dict]): A list of dictionaries to process.
            is_build (bool): Whether the operation is being executed in the build phase. Defaults to False.

        Returns:
            tuple[list[dict], float]: A tuple containing the filtered list of dictionaries
            and the total cost of the operation.

        This method performs the following steps:
        1. Processes each input item using an LLM model
        2. Validates the output
        3. Filters the results based on the specified filter key
        4. Calculates the total cost of the operation

        The method uses multi-threading to process items in parallel, improving performance
        for large datasets.

        Usage:
```

```
        ```python
        from docetl.operations import FilterOperation

        config = {
            "prompt": "Determine if the following item is important:
{{input}}",
            "output": {
                "schema": {"is_important": "bool"}
            },
            "model": "gpt-3.5-turbo"
        }
        filter_op = FilterOperation(config)
        input_data = [
            {"id": 1, "text": "Critical update"},
            {"id": 2, "text": "Regular maintenance"}
        ]
        results, cost = filter_op.execute(input_data)
        print(f"Filtered results: {results}")
        print(f"Total cost: {cost}")
        ```
        """
        filter_key = next(
            iter(
                [
                    k
                    for k in self.config["output"]["schema"].keys()
                    if k != "_short_explanation"
                ]
)
)

        results, total_cost = super().execute(input_data)

        # Drop records with filter_key values that are False
        if not is_build:
            results = [result for result in results if
result[filter_key]]

        # Drop the filter_key from the results
        for result in results:
            result.pop(filter_key, None)

        return results, total_cost
```

`execute(input_data, is_build=False)`

Executes the filter operation on the input data.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	A list of dictionaries to process.	*required*

Name	Type	Description	Default
`is_build`	`bool`	Whether the operation is being executed in the build phase. Defaults to False.	`False`

Returns:

Type	Description
`list[dict]`	tuple[list[dict], float]: A tuple containing the filtered list of dictionaries
`float`	and the total cost of the operation.

This method performs the following steps: 1. Processes each input item using an LLM model 2. Validates the output 3. Filters the results based on the specified filter key 4. Calculates the total cost of the operation

The method uses multi-threading to process items in parallel, improving performance for large datasets.

Usage:

```
from docetl.operations import FilterOperation

config = {
    "prompt": "Determine if the following item is important: {{input}}",
    "output": {
        "schema": {"is_important": "bool"}
    },
    "model": "gpt-3.5-turbo"
}
filter_op = FilterOperation(config)
input_data = [
    {"id": 1, "text": "Critical update"},
    {"id": 2, "text": "Regular maintenance"}
]
results, cost = filter_op.execute(input_data)
print(f"Filtered results: {results}")
print(f"Total cost: {cost}")
```

Source code in `docetl/operations/filter.py`

```python
def execute(
    self, input_data: list[dict], is_build: bool = False
) -> tuple[list[dict], float]:
    """
    Executes the filter operation on the input data.

    Args:
        input_data (list[dict]): A list of dictionaries to process.
        is_build (bool): Whether the operation is being executed in the
build phase. Defaults to False.

    Returns:
        tuple[list[dict], float]: A tuple containing the filtered list of
dictionaries
        and the total cost of the operation.

    This method performs the following steps:
    1. Processes each input item using an LLM model
    2. Validates the output
    3. Filters the results based on the specified filter key
    4. Calculates the total cost of the operation

    The method uses multi-threading to process items in parallel,
improving performance
    for large datasets.

    Usage:
    ```python
    from docetl.operations import FilterOperation

    config = {
        "prompt": "Determine if the following item is important:
{{input}}",
        "output": {
            "schema": {"is_important": "bool"}
        },
        "model": "gpt-3.5-turbo"
    }
    filter_op = FilterOperation(config)
    input_data = [
        {"id": 1, "text": "Critical update"},
        {"id": 2, "text": "Regular maintenance"}
    ]
    results, cost = filter_op.execute(input_data)
    print(f"Filtered results: {results}")
    print(f"Total cost: {cost}")
    ```
    """
    filter_key = next(
        iter(
            [
                k
                for k in self.config["output"]["schema"].keys()
                if k != "_short_explanation"
            ]
)
)
```

```
93
94      results, total_cost = super().execute(input_data)
95
96      # Drop records with filter_key values that are False
97      if not is_build:
98          results = [result for result in results if result[filter_key]]
99
100     # Drop the filter_key from the results
        for result in results:
            result.pop(filter_key, None)

        return results, total_cost
```

`docetl.operations.equijoin.EquijoinOperation`

Bases: `BaseOperation`

Source code in `docetl/operations/equijoin.py`

```python
class EquijoinOperation(BaseOperation):
    class schema(BaseOperation.schema):
        type: str = "equijoin"
        comparison_prompt: str
        output: dict[str, Any] | None = None
        blocking_threshold: float | None = None
        blocking_conditions: list[str] | None = None
        limits: dict[str, int] | None = None
        comparison_model: str | None = None
        optimize: bool | None = None
        embedding_model: str | None = None
        embedding_batch_size: int | None = None
        compare_batch_size: int | None = None
        limit_comparisons: int | None = None
        blocking_keys: dict[str, list[str]] | None = None
        timeout: int | None = None
        litellm_completion_kwargs: dict[str, Any] = {}

        @field_validator("blocking_keys")
        def validate_blocking_keys(cls, v):
            if v is not None:
                if "left" not in v or "right" not in v:
                    raise ValueError(
                        "Both 'left' and 'right' must be specified in
'blocking_keys'"
                    )
            return v

        @field_validator("limits")
        def validate_limits(cls, v):
            if v is not None:
                if "left" not in v or "right" not in v:
                    raise ValueError(
                        "Both 'left' and 'right' must be specified in
'limits'"
                    )
            return v

    def compare_pair(
        self,
        comparison_prompt: str,
        model: str,
        item1: dict,
        item2: dict,
        timeout_seconds: int = 120,
        max_retries_per_timeout: int = 2,
    ) -> tuple[bool, float]:
        """
        Compares two items using an LLM model to determine if they match.

        Args:
            comparison_prompt (str): The prompt template for comparison.
            model (str): The LLM model to use for comparison.
            item1 (dict): The first item to compare.
            item2 (dict): The second item to compare.
            timeout_seconds (int): The timeout for the LLM call in
seconds.
```

```
            max_retries_per_timeout (int): The maximum number of retries
per timeout.

        Returns:
            tuple[bool, float]: A tuple containing a boolean indicating
whether the items match and the cost of the comparison.
        """

        try:
            prompt = strict_render(comparison_prompt, {"left": item1,
"right": item2})
        except Exception as e:
            self.console.log(f"[red]Error rendering prompt: {e}[/red]")
            return False, 0
        response = self.runner.api.call_llm(
            model,
            "compare",
            [{"role": "user", "content": prompt}],
            {"is_match": "bool"},
            timeout_seconds=timeout_seconds,
            max_retries_per_timeout=max_retries_per_timeout,
            bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),

litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
{}),
            op_config=self.config,
        )
        cost = 0
        try:
            cost = response.total_cost
            output = self.runner.api.parse_llm_response(
                response.response, {"is_match": "bool"}
            )[0]
        except Exception as e:
            self.console.log(f"[red]Error parsing LLM response: {e}
[/red]")
            return False, cost
        return output["is_match"], cost

    def execute(
        self, left_data: list[dict], right_data: list[dict]
    ) -> tuple[list[dict], float]:
        """
        Executes the equijoin operation on the provided datasets.

        Args:
            left_data (list[dict]): The left dataset to join.
            right_data (list[dict]): The right dataset to join.

        Returns:
            tuple[list[dict], float]: A tuple containing the joined
results and the total cost of the operation.

        Usage:
        ```python
        from docetl.operations import EquijoinOperation

        config = {
            "blocking_keys": {
                "left": ["id"],
```
```

```
                "right": ["user_id"]
            },
            "limits": {
                "left": 1,
                "right": 1
            },
            "comparison_prompt": "Compare {{left}} and {{right}} and
determine if they match.",
            "blocking_threshold": 0.8,
            "blocking_conditions": ["left['id'] == right['user_id']"],
            "limit_comparisons": 1000
        }
        equijoin_op = EquijoinOperation(config)
        left_data = [{"id": 1, "name": "Alice"}, {"id": 2, "name":
"Bob"}]
        right_data = [{"user_id": 1, "age": 30}, {"user_id": 2, "age":
25}]
        results, cost = equijoin_op.execute(left_data, right_data)
        print(f"Joined results: {results}")
        print(f"Total cost: {cost}")
        ```

        This method performs the following steps:
        1. Initial blocking based on specified conditions (if any)
        2. Embedding-based blocking (if threshold is provided)
        3. LLM-based comparison for blocked pairs
        4. Result aggregation and validation

        The method also calculates and logs statistics such as
comparisons saved by blocking and join selectivity.
        """

        blocking_keys = self.config.get("blocking_keys", {})
        left_keys = blocking_keys.get(
            "left", list(left_data[0].keys()) if left_data else []
)
        right_keys = blocking_keys.get(
            "right", list(right_data[0].keys()) if right_data else []
)
        limits = self.config.get(
            "limits", {"left": float("inf"), "right": float("inf")}
)
        left_limit = limits["left"]
        right_limit = limits["right"]
        blocking_threshold = self.config.get("blocking_threshold")
        blocking_conditions = self.config.get("blocking_conditions", [])
        limit_comparisons = self.config.get("limit_comparisons")
        total_cost = 0

        if len(left_data) == 0 or len(right_data) == 0:
            return [], 0

        if self.status:
            self.status.stop()

        # Initial blocking using multiprocessing
        num_processes = min(cpu_count(), len(left_data))

        self.console.log(
            f"Starting to run code-based blocking rules for
{len(left_data)} left and {len(right_data)} right rows ({len(left_data) *
```
```

```
len(right_data)} total pairs) with {num_processes} processes..."
        )

        with Pool(
            processes=num_processes,
            initializer=init_worker,
            initargs=(right_data, blocking_conditions),
        ) as pool:
            blocked_pairs_nested = pool.map(process_left_item, left_data)

        # Flatten the nested list of blocked pairs
        blocked_pairs = [pair for sublist in blocked_pairs_nested for
pair in sublist]

        # Check if we have exceeded the pairwise comparison limit
        if limit_comparisons is not None and len(blocked_pairs) >
limit_comparisons:
            # Sample pairs based on cardinality and length
            sampled_pairs = stratified_length_sample(
                blocked_pairs, limit_comparisons, sample_size=1000,
console=self.console
            )

            # Calculate number of dropped pairs
            dropped_pairs = len(blocked_pairs) - limit_comparisons

            # Prompt the user for confirmation
            if self.status:
                self.status.stop()
            if not Confirm.ask(
                f"[yellow]Warning: {dropped_pairs} pairs will be dropped
due to the comparison limit. "
                f"Proceeding with {limit_comparisons} randomly sampled
pairs. "
                f"Do you want to continue?[/yellow]",
                console=self.console,
            ):
                raise ValueError("Operation cancelled by user due to pair
limit.")

            if self.status:
                self.status.start()

            blocked_pairs = sampled_pairs

        self.console.log(
            f"Number of blocked pairs after initial blocking:
{len(blocked_pairs)}"
        )

        if blocking_threshold is not None:
            embedding_model = self.config.get("embedding_model",
self.default_model)
            model_input_context_length = model_cost.get(embedding_model,
{}).get(
                "max_input_tokens", 8192
            )

            def get_embeddings(
                input_data: list[dict[str, Any]], keys: list[str], name:
str
```

```
            ) -> tuple[list[list[float]], float]:
                texts = [
                    " ".join(str(item[key]) for key in keys if key in
item)[
                        : model_input_context_length * 4
                    ]
                    for item in input_data
                ]

                embeddings = []
                total_cost = 0
                batch_size = 2000
                for i in range(0, len(texts), batch_size):
                    batch = texts[i : i + batch_size]
                    self.console.log(
                        f"On iteration {i} for creating embeddings for
{name} data"
                    )
                    response = self.runner.api.gen_embedding(
                        model=embedding_model,
                        input=batch,
                    )
                    embeddings.extend([data["embedding"] for data in
response["data"]])
                    total_cost += completion_cost(response)
                return embeddings, total_cost

            left_embeddings, left_cost = get_embeddings(left_data,
left_keys, "left")
            right_embeddings, right_cost = get_embeddings(
                right_data, right_keys, "right"
            )
            total_cost += left_cost + right_cost
            self.console.log(
                f"Created embeddings for datasets. Total embedding
creation cost: {total_cost}"
            )

            # Compute all cosine similarities in one call
            from sklearn.metrics.pairwise import cosine_similarity

            similarities = cosine_similarity(left_embeddings,
right_embeddings)

            # Additional blocking based on embeddings
            # Find indices where similarity is above threshold
            above_threshold = np.argwhere(similarities >=
blocking_threshold)
            self.console.log(
                f"There are {above_threshold.shape[0]} pairs above the
threshold."
            )
            block_pair_set = set(
                (get_hashable_key(left_item),
get_hashable_key(right_item))
                for left_item, right_item in blocked_pairs
            )

            # If limit_comparisons is set, take only the top pairs
            if limit_comparisons is not None:
                # First, get all pairs above threshold
```

```
                above_threshold_pairs = [(int(i), int(j)) for i, j in
above_threshold]

                # Sort these pairs by their similarity scores
                sorted_pairs = sorted(
                    above_threshold_pairs,
                    key=lambda pair: similarities[pair[0], pair[1]],
                    reverse=True,
                )

                # Take the top 'limit_comparisons' pairs
                top_pairs = sorted_pairs[:limit_comparisons]

                # Create new blocked_pairs based on top similarities and
existing blocked pairs
                new_blocked_pairs = []
                remaining_limit = limit_comparisons - len(blocked_pairs)

                # First, include all existing blocked pairs
                final_blocked_pairs = blocked_pairs.copy()

                # Then, add new pairs from top similarities until we
reach the limit
                for i, j in top_pairs:
                    if remaining_limit <= 0:
                        break
                    left_item, right_item = left_data[i], right_data[j]
                    left_key = get_hashable_key(left_item)
                    right_key = get_hashable_key(right_item)
                    if (left_key, right_key) not in block_pair_set:
                        new_blocked_pairs.append((left_item, right_item))
                        block_pair_set.add((left_key, right_key))
                        remaining_limit -= 1

                final_blocked_pairs.extend(new_blocked_pairs)
                blocked_pairs = final_blocked_pairs

                self.console.log(
                    f"Limited comparisons to top {limit_comparisons}
pairs, including {len(blocked_pairs) - len(new_blocked_pairs)} from code-
based blocking and {len(new_blocked_pairs)} based on cosine similarity.
Lowest cosine similarity included: {similarities[top_pairs[-1]]:.4f}"
                )
            else:
                # Add new pairs to blocked_pairs
                for i, j in above_threshold:
                    left_item, right_item = left_data[i], right_data[j]
                    left_key = get_hashable_key(left_item)
                    right_key = get_hashable_key(right_item)
                    if (left_key, right_key) not in block_pair_set:
                        blocked_pairs.append((left_item, right_item))
                        block_pair_set.add((left_key, right_key))

        # If there are no blocking conditions or embedding threshold, use
all pairs
        if not blocking_conditions and blocking_threshold is None:
            blocked_pairs = [
                (left_item, right_item)
                for left_item in left_data
                for right_item in right_data
            ]
```

```
418
419         # If there's a limit on the number of comparisons, randomly
420 sample pairs
421         if limit_comparisons is not None and len(blocked_pairs) >
422 limit_comparisons:
423             self.console.log(
424                 f"Randomly sampling {limit_comparisons} pairs out of
425 {len(blocked_pairs)} blocked pairs."
426             )
427             blocked_pairs = random.sample(blocked_pairs,
428 limit_comparisons)
429
430         self.console.log(
431             f"Total pairs to compare after blocking and sampling:
432 {len(blocked_pairs)}"
433         )
434
435         # Calculate and print statistics
436         total_possible_comparisons = len(left_data) * len(right_data)
437         comparisons_made = len(blocked_pairs)
438         comparisons_saved = total_possible_comparisons - comparisons_made
439         self.console.log(
440             f"[green]Comparisons saved by blocking: {comparisons_saved} "
441             f"({(comparisons_saved / total_possible_comparisons) *
442 100:.2f}%)[/green]"
443         )
444
445         left_match_counts = defaultdict(int)
446         right_match_counts = defaultdict(int)
447         results = []
448         comparison_costs = 0
449
450         if self.status:
451             self.status.stop()
452
453         with ThreadPoolExecutor(max_workers=self.max_threads) as
454 executor:
455             future_to_pair = {
456                 executor.submit(
457                     self.compare_pair,
458                     self.config["comparison_prompt"],
459                     self.config.get("comparison_model",
460 self.default_model),
461                     left,
462                     right,
463                     self.config.get("timeout", 120),
464                     self.config.get("max_retries_per_timeout", 2),
465                 ): (left, right)
466                 for left, right in blocked_pairs
467             }
468
469             pbar = RichLoopBar(
470                 range(len(future_to_pair)),
                 desc="Comparing pairs",
                 console=self.console,
             )

             for i in pbar:
                 future = list(future_to_pair.keys())[i]
                 pair = future_to_pair[future]
                 is_match, cost = future.result()
```

```
                    comparison_costs += cost

                    if is_match:
                        joined_item = {}
                        left_item, right_item = pair
                        left_key_hash = get_hashable_key(left_item)
                        right_key_hash = get_hashable_key(right_item)
                        if (
                            left_match_counts[left_key_hash] >= left_limit
                            or right_match_counts[right_key_hash] >=
    right_limit
                        ):
                            continue

                        for key, value in left_item.items():
                            joined_item[f"{key}_left" if key in right_item
    else key] = value
                        for key, value in right_item.items():
                            joined_item[f"{key}_right" if key in left_item
    else key] = value
                        if self.runner.api.validate_output(
                            self.config, joined_item, self.console
                        ):
                            results.append(joined_item)
                            left_match_counts[left_key_hash] += 1
                            right_match_counts[right_key_hash] += 1

                        # TODO: support retry in validation failure

            total_cost += comparison_costs

            if self.status:
                self.status.start()

            # Calculate and print the join selectivity
            join_selectivity = (
                len(results) / (len(left_data) * len(right_data))
                if len(left_data) * len(right_data) > 0
                else 0
            )
            self.console.log(f"Equijoin selectivity: {join_selectivity:.4f}")

            if self.status:
                self.status.start()

            return results, total_cost
```

`compare_pair(comparison_prompt, model, item1, item2, timeout_seconds=120, max_retries_per_timeout=2)`

Compares two items using an LLM model to determine if they match.

Parameters:

Name	Type	Description	Default
`comparison_prompt`	`str`	The prompt template for comparison.	*required*
`model`	`str`	The LLM model to use for comparison.	*required*
`item1`	`dict`	The first item to compare.	*required*
`item2`	`dict`	The second item to compare.	*required*
`timeout_seconds`	`int`	The timeout for the LLM call in seconds.	`120`
`max_retries_per_timeout`	`int`	The maximum number of retries per timeout.	`2`

Returns:

Type	Description
`tuple[bool, float]`	tuple[bool, float]: A tuple containing a boolean indicating whether the items match and the cost of the comparison.

Source code in `docetl/operations/equijoin.py`

```python
def compare_pair(
    self,
    comparison_prompt: str,
    model: str,
    item1: dict,
    item2: dict,
    timeout_seconds: int = 120,
    max_retries_per_timeout: int = 2,
) -> tuple[bool, float]:
    """
    Compares two items using an LLM model to determine if they match.

    Args:
        comparison_prompt (str): The prompt template for comparison.
        model (str): The LLM model to use for comparison.
        item1 (dict): The first item to compare.
        item2 (dict): The second item to compare.
        timeout_seconds (int): The timeout for the LLM call in seconds.
        max_retries_per_timeout (int): The maximum number of retries per
timeout.

    Returns:
        tuple[bool, float]: A tuple containing a boolean indicating
whether the items match and the cost of the comparison.
    """

    try:
        prompt = strict_render(comparison_prompt, {"left": item1,
"right": item2})
    except Exception as e:
        self.console.log(f"[red]Error rendering prompt: {e}[/red]")
        return False, 0
    response = self.runner.api.call_llm(
        model,
        "compare",
        [{"role": "user", "content": prompt}],
        {"is_match": "bool"},
        timeout_seconds=timeout_seconds,
        max_retries_per_timeout=max_retries_per_timeout,
        bypass_cache=self.config.get("bypass_cache", self.bypass_cache),

litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
{}),
        op_config=self.config,
    )
    cost = 0
    try:
        cost = response.total_cost
        output = self.runner.api.parse_llm_response(
            response.response, {"is_match": "bool"}
        )[0]
    except Exception as e:
        self.console.log(f"[red]Error parsing LLM response: {e}[/red]")
        return False, cost
    return output["is_match"], cost
```

`execute(left_data, right_data)`

Executes the equijoin operation on the provided datasets.

Parameters:

Name	Type	Description	Default
`left_data`	`list[dict]`	The left dataset to join.	*required*
`right_data`	`list[dict]`	The right dataset to join.	*required*

Returns:

Type	Description
`tuple[list[dict], float]`	tuple[list[dict], float]: A tuple containing the joined results and the total cost of the operation.

Usage:

```
from docetl.operations import EquijoinOperation

config = {
    "blocking_keys": {
        "left": ["id"],
        "right": ["user_id"]
    },
    "limits": {
        "left": 1,
        "right": 1
    },
    "comparison_prompt": "Compare {{left}} and {{right}} and determine if they
match.",
    "blocking_threshold": 0.8,
    "blocking_conditions": ["left['id'] == right['user_id']"],
    "limit_comparisons": 1000
}
equijoin_op = EquijoinOperation(config)
left_data = [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]
right_data = [{"user_id": 1, "age": 30}, {"user_id": 2, "age": 25}]
results, cost = equijoin_op.execute(left_data, right_data)
print(f"Joined results: {results}")
print(f"Total cost: {cost}")
```

This method performs the following steps: 1. Initial blocking based on specified conditions (if any) 2. Embedding-based blocking (if threshold is provided) 3. LLM-based

comparison for blocked pairs 4. Result aggregation and validation

The method also calculates and logs statistics such as comparisons saved by blocking and join selectivity.

Source code in `docetl/operations/equijoin.py`

```
def execute(
    self, left_data: list[dict], right_data: list[dict]
) -> tuple[list[dict], float]:
    """
    Executes the equijoin operation on the provided datasets.

    Args:
        left_data (list[dict]): The left dataset to join.
        right_data (list[dict]): The right dataset to join.

    Returns:
        tuple[list[dict], float]: A tuple containing the joined results
and the total cost of the operation.

    Usage:
    ```python
    from docetl.operations import EquijoinOperation

    config = {
        "blocking_keys": {
            "left": ["id"],
            "right": ["user_id"]
        },
        "limits": {
            "left": 1,
            "right": 1
        },
        "comparison_prompt": "Compare {{left}} and {{right}} and
determine if they match.",
        "blocking_threshold": 0.8,
        "blocking_conditions": ["left['id'] == right['user_id']"],
        "limit_comparisons": 1000
    }
    equijoin_op = EquijoinOperation(config)
    left_data = [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]
    right_data = [{"user_id": 1, "age": 30}, {"user_id": 2, "age": 25}]
    results, cost = equijoin_op.execute(left_data, right_data)
    print(f"Joined results: {results}")
    print(f"Total cost: {cost}")
    ```

    This method performs the following steps:
    1. Initial blocking based on specified conditions (if any)
    2. Embedding-based blocking (if threshold is provided)
    3. LLM-based comparison for blocked pairs
    4. Result aggregation and validation

    The method also calculates and logs statistics such as comparisons
saved by blocking and join selectivity.
    """

    blocking_keys = self.config.get("blocking_keys", {})
    left_keys = blocking_keys.get(
        "left", list(left_data[0].keys()) if left_data else []
)
    right_keys = blocking_keys.get(
        "right", list(right_data[0].keys()) if right_data else []
```

```
    )
    limits = self.config.get(
        "limits", {"left": float("inf"), "right": float("inf")}
    )
    left_limit = limits["left"]
    right_limit = limits["right"]
    blocking_threshold = self.config.get("blocking_threshold")
    blocking_conditions = self.config.get("blocking_conditions", [])
    limit_comparisons = self.config.get("limit_comparisons")
    total_cost = 0

    if len(left_data) == 0 or len(right_data) == 0:
        return [], 0

    if self.status:
        self.status.stop()

    # Initial blocking using multiprocessing
    num_processes = min(cpu_count(), len(left_data))

    self.console.log(
        f"Starting to run code-based blocking rules for {len(left_data)}
left and {len(right_data)} right rows ({len(left_data) * len(right_data)}
total pairs) with {num_processes} processes..."
    )

    with Pool(
        processes=num_processes,
        initializer=init_worker,
        initargs=(right_data, blocking_conditions),
    ) as pool:
        blocked_pairs_nested = pool.map(process_left_item, left_data)

    # Flatten the nested list of blocked pairs
    blocked_pairs = [pair for sublist in blocked_pairs_nested for pair in
sublist]

    # Check if we have exceeded the pairwise comparison limit
    if limit_comparisons is not None and len(blocked_pairs) >
limit_comparisons:
        # Sample pairs based on cardinality and length
        sampled_pairs = stratified_length_sample(
            blocked_pairs, limit_comparisons, sample_size=1000,
console=self.console
        )

        # Calculate number of dropped pairs
        dropped_pairs = len(blocked_pairs) - limit_comparisons

        # Prompt the user for confirmation
        if self.status:
            self.status.stop()
        if not Confirm.ask(
            f"[yellow]Warning: {dropped_pairs} pairs will be dropped due
to the comparison limit. "
            f"Proceeding with {limit_comparisons} randomly sampled pairs.
"
            f"Do you want to continue?[/yellow]",
            console=self.console,
        ):
            raise ValueError("Operation cancelled by user due to pair
```

```
limit.")

        if self.status:
            self.status.start()

        blocked_pairs = sampled_pairs

    self.console.log(
        f"Number of blocked pairs after initial blocking:
{len(blocked_pairs)}"
    )

    if blocking_threshold is not None:
        embedding_model = self.config.get("embedding_model",
self.default_model)
        model_input_context_length = model_cost.get(embedding_model,
{}).get(
            "max_input_tokens", 8192
        )

        def get_embeddings(
            input_data: list[dict[str, Any]], keys: list[str], name: str
        ) -> tuple[list[list[float]], float]:
            texts = [
                " ".join(str(item[key]) for key in keys if key in item)[
                    : model_input_context_length * 4
                ]
                for item in input_data
            ]

            embeddings = []
            total_cost = 0
            batch_size = 2000
            for i in range(0, len(texts), batch_size):
                batch = texts[i : i + batch_size]
                self.console.log(
                    f"On iteration {i} for creating embeddings for {name}
data"
                )
                response = self.runner.api.gen_embedding(
                    model=embedding_model,
                    input=batch,
                )
                embeddings.extend([data["embedding"] for data in
response["data"]])
                total_cost += completion_cost(response)
            return embeddings, total_cost

        left_embeddings, left_cost = get_embeddings(left_data, left_keys,
"left")
        right_embeddings, right_cost = get_embeddings(
            right_data, right_keys, "right"
        )
        total_cost += left_cost + right_cost
        self.console.log(
            f"Created embeddings for datasets. Total embedding creation
cost: {total_cost}"
        )

        # Compute all cosine similarities in one call
        from sklearn.metrics.pairwise import cosine_similarity
```

```
322
323         similarities = cosine_similarity(left_embeddings,
324 right_embeddings)
325
326         # Additional blocking based on embeddings
327         # Find indices where similarity is above threshold
328         above_threshold = np.argwhere(similarities >= blocking_threshold)
329         self.console.log(
330             f"There are {above_threshold.shape[0]} pairs above the
331 threshold."
332         )
333         block_pair_set = set(
334             (get_hashable_key(left_item), get_hashable_key(right_item))
335             for left_item, right_item in blocked_pairs
336         )
337
338         # If limit_comparisons is set, take only the top pairs
339         if limit_comparisons is not None:
340             # First, get all pairs above threshold
341             above_threshold_pairs = [(int(i), int(j)) for i, j in
342 above_threshold]
343
344             # Sort these pairs by their similarity scores
345             sorted_pairs = sorted(
346                 above_threshold_pairs,
347                 key=lambda pair: similarities[pair[0], pair[1]],
348                 reverse=True,
349             )
350
351             # Take the top 'limit_comparisons' pairs
352             top_pairs = sorted_pairs[:limit_comparisons]
353
354             # Create new blocked_pairs based on top similarities and
355 existing blocked pairs
356             new_blocked_pairs = []
357             remaining_limit = limit_comparisons - len(blocked_pairs)
358
359             # First, include all existing blocked pairs
360             final_blocked_pairs = blocked_pairs.copy()
361
362             # Then, add new pairs from top similarities until we reach
363 the limit
364             for i, j in top_pairs:
365                 if remaining_limit <= 0:
366                     break
367                 left_item, right_item = left_data[i], right_data[j]
368                 left_key = get_hashable_key(left_item)
369                 right_key = get_hashable_key(right_item)
370                 if (left_key, right_key) not in block_pair_set:
371                     new_blocked_pairs.append((left_item, right_item))
372                     block_pair_set.add((left_key, right_key))
373                     remaining_limit -= 1
374
375             final_blocked_pairs.extend(new_blocked_pairs)
376             blocked_pairs = final_blocked_pairs
377
378             self.console.log(
379                 f"Limited comparisons to top {limit_comparisons} pairs,
380 including {len(blocked_pairs) - len(new_blocked_pairs)} from code-based
381 blocking and {len(new_blocked_pairs)} based on cosine similarity. Lowest
382 cosine similarity included: {similarities[top_pairs[-1]]:.4f}"
```

```
            )
        else:
            # Add new pairs to blocked_pairs
            for i, j in above_threshold:
                left_item, right_item = left_data[i], right_data[j]
                left_key = get_hashable_key(left_item)
                right_key = get_hashable_key(right_item)
                if (left_key, right_key) not in block_pair_set:
                    blocked_pairs.append((left_item, right_item))
                    block_pair_set.add((left_key, right_key))

    # If there are no blocking conditions or embedding threshold, use all
pairs
    if not blocking_conditions and blocking_threshold is None:
        blocked_pairs = [
            (left_item, right_item)
            for left_item in left_data
            for right_item in right_data
        ]

    # If there's a limit on the number of comparisons, randomly sample
pairs
    if limit_comparisons is not None and len(blocked_pairs) >
limit_comparisons:
        self.console.log(
            f"Randomly sampling {limit_comparisons} pairs out of
{len(blocked_pairs)} blocked pairs."
        )
        blocked_pairs = random.sample(blocked_pairs, limit_comparisons)

    self.console.log(
        f"Total pairs to compare after blocking and sampling:
{len(blocked_pairs)}"
    )

    # Calculate and print statistics
    total_possible_comparisons = len(left_data) * len(right_data)
    comparisons_made = len(blocked_pairs)
    comparisons_saved = total_possible_comparisons - comparisons_made
    self.console.log(
        f"[green]Comparisons saved by blocking: {comparisons_saved} "
        f"({(comparisons_saved / total_possible_comparisons) * 100:.2f}%)
[/green]"
    )

    left_match_counts = defaultdict(int)
    right_match_counts = defaultdict(int)
    results = []
    comparison_costs = 0

    if self.status:
        self.status.stop()

    with ThreadPoolExecutor(max_workers=self.max_threads) as executor:
        future_to_pair = {
            executor.submit(
                self.compare_pair,
                self.config["comparison_prompt"],
                self.config.get("comparison_model", self.default_model),
                left,
                right,
```

```
                self.config.get("timeout", 120),
                self.config.get("max_retries_per_timeout", 2),
            ): (left, right)
            for left, right in blocked_pairs
        }

        pbar = RichLoopBar(
            range(len(future_to_pair)),
            desc="Comparing pairs",
            console=self.console,
        )

        for i in pbar:
            future = list(future_to_pair.keys())[i]
            pair = future_to_pair[future]
            is_match, cost = future.result()
            comparison_costs += cost

            if is_match:
                joined_item = {}
                left_item, right_item = pair
                left_key_hash = get_hashable_key(left_item)
                right_key_hash = get_hashable_key(right_item)
                if (
                    left_match_counts[left_key_hash] >= left_limit
                    or right_match_counts[right_key_hash] >= right_limit
                ):
                    continue

                for key, value in left_item.items():
                    joined_item[f"{key}_left" if key in right_item else
key] = value
                for key, value in right_item.items():
                    joined_item[f"{key}_right" if key in left_item else
key] = value
                if self.runner.api.validate_output(
                    self.config, joined_item, self.console
                ):
                    results.append(joined_item)
                    left_match_counts[left_key_hash] += 1
                    right_match_counts[right_key_hash] += 1

                # TODO: support retry in validation failure

    total_cost += comparison_costs

    if self.status:
        self.status.start()

    # Calculate and print the join selectivity
    join_selectivity = (
        len(results) / (len(left_data) * len(right_data))
        if len(left_data) * len(right_data) > 0
        else 0
    )
    self.console.log(f"Equijoin selectivity: {join_selectivity:.4f}")

    if self.status:
        self.status.start()
```

```
        return results, total_cost
```

`docetl.operations.cluster.ClusterOperation`

Bases: `BaseOperation`

Source code in `docetl/operations/cluster.py`

```python
class ClusterOperation(BaseOperation):
    def __init__(
        self,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)
        self.max_batch_size: int = self.config.get(
            "max_batch_size", kwargs.get("max_batch_size", float("inf"))
        )

    def syntax_check(self) -> None:
        """
        Checks the configuration of the ClusterOperation for required
keys and valid structure.

        Raises:
            ValueError: If required keys are missing or invalid in the
configuration.
            TypeError: If configuration values have incorrect types.
        """
        required_keys = ["embedding_keys", "summary_schema",
"summary_prompt"]
        for key in required_keys:
            if key not in self.config:
                raise ValueError(
                    f"Missing required key '{key}' in ClusterOperation
configuration"
                )

        if not isinstance(self.config["embedding_keys"], list):
            raise TypeError("'embedding_keys' must be a list of strings")

        if "output_key" in self.config:
            if not isinstance(self.config["output_key"], str):
                raise TypeError("'output_key' must be a string")

        if not isinstance(self.config["summary_schema"], dict):
            raise TypeError("'summary_schema' must be a dictionary")

        if not isinstance(self.config["summary_prompt"], str):
            raise TypeError("'prompt' must be a string")

        # Check if the prompt is a valid Jinja2 template
        try:
            Template(self.config["summary_prompt"])
        except Exception as e:
            raise ValueError(f"Invalid Jinja2 template in 'prompt':
{str(e)}")

        # Check optional parameters
        if "max_batch_size" in self.config:
            if not isinstance(self.config["max_batch_size"], int):
                raise TypeError("'max_batch_size' must be an integer")

        if "embedding_model" in self.config:
            if not isinstance(self.config["embedding_model"], str):
```

```
                raise TypeError("'embedding_model' must be a string")

        if "model" in self.config:
            if not isinstance(self.config["model"], str):
                raise TypeError("'model' must be a string")

        if "validate" in self.config:
            if not isinstance(self.config["validate"], list):
                raise TypeError("'validate' must be a list of strings")
            for rule in self.config["validate"]:
                if not isinstance(rule, str):
                    raise TypeError("Each validation rule must be a
string")

    def execute(
        self, input_data: list[dict], is_build: bool = False
    ) -> tuple[list[dict], float]:
        """
        Executes the cluster operation on the input data. Modifies the
        input data and returns it in place.

        Args:
            input_data (list[dict]): A list of dictionaries to process.
            is_build (bool): Whether the operation is being executed
              in the build phase. Defaults to False.

        Returns:
            tuple[list[dict], float]: A tuple containing the clustered
              list of dictionaries and the total cost of the operation.
        """
        if not input_data:
            return input_data, 0

        if len(input_data) == 1:
            input_data[0][self.config.get("output_key", "clusters")] = ()
            return input_data, 0

        embeddings, cost = get_embeddings_for_clustering(
            input_data, self.config, self.runner.api
        )

        tree = self.agglomerative_cluster_of_embeddings(input_data,
embeddings)

        if "collapse" in self.config:
            tree = self.collapse_tree(tree,
collapse=self.config["collapse"])

        self.prompt_template = Template(self.config["summary_prompt"])
        cost += self.annotate_clustering_tree(tree)
        self.annotate_leaves(tree)

        return input_data, cost

    def agglomerative_cluster_of_embeddings(self, input_data,
embeddings):
        import sklearn.cluster

        cl = sklearn.cluster.AgglomerativeClustering(
            compute_full_tree=True, compute_distances=True
        )
```

```
        cl.fit(embeddings)

        nsamples = len(embeddings)

        def build_tree(i):
            if i < nsamples:
                res = input_data[i]
                #                res["embedding"] = list(embeddings[i])
                return res
            return {
                "children": [
                    build_tree(cl.children_[i - nsamples, 0]),
                    build_tree(cl.children_[i - nsamples, 1]),
                ],
                "distance": cl.distances_[i - nsamples],
            }

        return build_tree(nsamples + len(cl.children_) - 1)

    def get_tree_distances(self, t):
        res = set()
        if "distance" in t:
            res.update(
                set(
                    [
                        t["distance"] - child["distance"]
                        for child in t["children"]
                        if "distance" in child
                    ]
                )
            )
        if "children" in t:
            for child in t["children"]:
                res.update(self.get_tree_distances(child))
        return res

    def _collapse_tree(self, t, parent_dist=None, collapse=None):
        if "children" in t:
            if (
                "distance" in t
                and parent_dist is not None
                and collapse is not None
                and parent_dist - t["distance"] < collapse
            ):
                return [
                    grandchild
                    for child in t["children"]
                    for grandchild in self._collapse_tree(
                        child, parent_dist=parent_dist, collapse=collapse
                    )
                ]
            else:
                res = dict(t)
                res["children"] = [
                    grandchild
                    for idx, child in enumerate(t["children"])
                    for grandchild in self._collapse_tree(
                        child, parent_dist=t["distance"],
collapse=collapse
                    )
                ]
```

```
                return [res]
        else:
            return [t]

    def collapse_tree(self, tree, collapse=None):
        if collapse is not None:
            tree_distances =
np.array(sorted(self.get_tree_distances(tree)))
            collapse = tree_distances[int(len(tree_distances) *
collapse)]
        return self._collapse_tree(tree, collapse=collapse)[0]

    def annotate_clustering_tree(self, t):
        if "children" in t:
            with ThreadPoolExecutor(max_workers=self.max_batch_size) as
executor:
                futures = [
                    executor.submit(self.annotate_clustering_tree, child)
                    for child in t["children"]
                ]

                total_cost = 0
                pbar = RichLoopBar(
                    range(len(futures)),
                    desc=f"Processing {self.config['name']} (map) on all
documents",
                    console=self.console,
                )
                for i in pbar:
                    total_cost += futures[i].result()
                    pbar.update(i)

            prompt = strict_render(self.prompt_template, {"inputs":
t["children"]})

            def validation_fn(response: dict[str, Any]):
                output = self.runner.api.parse_llm_response(
                    response,
                    schema=self.config["summary_schema"],
                    manually_fix_errors=self.manually_fix_errors,
                )[0]
                if self.runner.api.validate_output(self.config, output,
self.console):
                    return output, True
                return output, False

            response = self.runner.api.call_llm(
                model=self.config.get("model", self.default_model),
                op_type="cluster",
                messages=[{"role": "user", "content": prompt}],
                output_schema=self.config["summary_schema"],
                timeout_seconds=self.config.get("timeout", 120),
                bypass_cache=self.config.get("bypass_cache",
self.bypass_cache),

max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
                validation_config=(
                    {
                        "num_retries":
self.num_retries_on_validate_failure,
                        "val_rule": self.config.get("validate", []),
```

```
                        "validation_fn": validation_fn,
                    }
                    if self.config.get("validate", None)
                    else None
                ),
                verbose=self.config.get("verbose", False),
                litellm_completion_kwargs=self.config.get(
                    "litellm_completion_kwargs", {}
                ),
                op_config=self.config,
            )
            total_cost += response.total_cost
            if response.validated:
                output = self.runner.api.parse_llm_response(
                    response.response,
                    schema=self.config["summary_schema"],
                    manually_fix_errors=self.manually_fix_errors,
                )[0]
                t.update(output)

            return total_cost
        return 0

    def annotate_leaves(self, tree, path=()):
        if "children" in tree:
            item = dict(tree)
            item.pop("children")
            for child in tree["children"]:
                self.annotate_leaves(child, path=(item,) + path)
        else:
            tree[self.config.get("output_key", "clusters")] = path
```

`execute(input_data, is_build=False)`

Executes the cluster operation on the input data. Modifies the input data and returns it in place.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	A list of dictionaries to process.	*required*
`is_build`	`bool`	Whether the operation is being executed in the build phase. Defaults to False.	`False`

Returns:

Type	Description
`tuple[list[dict], float]`	tuple[list[dict], float]: A tuple containing the clustered list of dictionaries and the total cost of the operation.

Source code in `docetl/operations/cluster.py`

```python
def execute(
    self, input_data: list[dict], is_build: bool = False
) -> tuple[list[dict], float]:
    """
    Executes the cluster operation on the input data. Modifies the
    input data and returns it in place.

    Args:
        input_data (list[dict]): A list of dictionaries to process.
        is_build (bool): Whether the operation is being executed
          in the build phase. Defaults to False.

    Returns:
        tuple[list[dict], float]: A tuple containing the clustered
          list of dictionaries and the total cost of the operation.
    """
    if not input_data:
        return input_data, 0

    if len(input_data) == 1:
        input_data[0][self.config.get("output_key", "clusters")] = ()
        return input_data, 0

    embeddings, cost = get_embeddings_for_clustering(
        input_data, self.config, self.runner.api
    )

    tree = self.agglomerative_cluster_of_embeddings(input_data,
embeddings)

    if "collapse" in self.config:
        tree = self.collapse_tree(tree, collapse=self.config["collapse"])

    self.prompt_template = Template(self.config["summary_prompt"])
    cost += self.annotate_clustering_tree(tree)
    self.annotate_leaves(tree)

    return input_data, cost
```

`syntax_check()`

Checks the configuration of the ClusterOperation for required keys and valid structure.

Raises:

Type	Description
`ValueError`	If required keys are missing or invalid in the configuration.
`TypeError`	If configuration values have incorrect types.

Source code in `docetl/operations/cluster.py`

```
def syntax_check(self) -> None:
    """
    Checks the configuration of the ClusterOperation for required keys and
valid structure.

    Raises:
        ValueError: If required keys are missing or invalid in the
configuration.
        TypeError: If configuration values have incorrect types.
    """
    required_keys = ["embedding_keys", "summary_schema", "summary_prompt"]
    for key in required_keys:
        if key not in self.config:
            raise ValueError(
                f"Missing required key '{key}' in ClusterOperation
configuration"
            )

    if not isinstance(self.config["embedding_keys"], list):
        raise TypeError("'embedding_keys' must be a list of strings")

    if "output_key" in self.config:
        if not isinstance(self.config["output_key"], str):
            raise TypeError("'output_key' must be a string")

    if not isinstance(self.config["summary_schema"], dict):
        raise TypeError("'summary_schema' must be a dictionary")

    if not isinstance(self.config["summary_prompt"], str):
        raise TypeError("'prompt' must be a string")

    # Check if the prompt is a valid Jinja2 template
    try:
        Template(self.config["summary_prompt"])
    except Exception as e:
        raise ValueError(f"Invalid Jinja2 template in 'prompt': {str(e)}")

    # Check optional parameters
    if "max_batch_size" in self.config:
        if not isinstance(self.config["max_batch_size"], int):
            raise TypeError("'max_batch_size' must be an integer")

    if "embedding_model" in self.config:
        if not isinstance(self.config["embedding_model"], str):
            raise TypeError("'embedding_model' must be a string")

    if "model" in self.config:
        if not isinstance(self.config["model"], str):
            raise TypeError("'model' must be a string")

    if "validate" in self.config:
        if not isinstance(self.config["validate"], list):
            raise TypeError("'validate' must be a list of strings")
        for rule in self.config["validate"]:
            if not isinstance(rule, str):
                raise TypeError("Each validation rule must be a string")
```

Auxiliary Operators

`docetl.operations.split.SplitOperation`

Bases: `BaseOperation`

A class that implements a split operation on input data, dividing it into manageable chunks.

This class extends BaseOperation to: 1. Split input data into chunks of specified size based on the 'split_key' and 'token_count' configuration. 2. Assign unique identifiers to each original document and number chunks sequentially. 3. Return results containing: - {split_key}_chunk: The content of the split chunk. - {name}_id: A unique identifier for each original document. - {name}_chunk_num: The sequential number of the chunk within its original document.

Source code in `docetl/operations/split.py`

```python
class SplitOperation(BaseOperation):
    """
    A class that implements a split operation on input data, dividing it into manageable chunks.

    This class extends BaseOperation to:
    1. Split input data into chunks of specified size based on the 'split_key' and 'token_count' configuration.
    2. Assign unique identifiers to each original document and number chunks sequentially.
    3. Return results containing:
       - {split_key}_chunk: The content of the split chunk.
       - {name}_id: A unique identifier for each original document.
       - {name}_chunk_num: The sequential number of the chunk within its original document.
    """

    class schema(BaseOperation.schema):
        type: str = "split"
        split_key: str
        method: str
        method_kwargs: dict[str, Any]
        model: str | None = None

        @field_validator("method")
        def validate_method(cls, v):
            if v not in ["token_count", "delimiter"]:
                raise ValueError(
                    f"Invalid method '{v}'. Must be 'token_count' or 'delimiter'"
                )
            return v

        @model_validator(mode="after")
        def validate_method_kwargs(self):
            if self.method == "token_count":
                num_tokens = self.method_kwargs.get("num_tokens")
                if num_tokens is None or num_tokens <= 0:
                    raise ValueError("'num_tokens' must be a positive integer")
            elif self.method == "delimiter":
                if "delimiter" not in self.method_kwargs:
                    raise ValueError("'delimiter' is required for delimiter method")
            return self

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.name = self.config["name"]

    def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
        split_key = self.config["split_key"]
        method = self.config["method"]
        method_kwargs = self.config["method_kwargs"]
        try:
            encoder = tiktoken.encoding_for_model(
```

```
                self.config["method_kwargs"]
                .get("model", self.default_model)
                .split("/")[-1]
            )
        except Exception:
            encoder = tiktoken.encoding_for_model("gpt-4o")

        results = []
        cost = 0.0

        for item in input_data:
            if split_key not in item:
                raise KeyError(f"Split key '{split_key}' not found in
item")

            content = item[split_key]
            doc_id = str(uuid.uuid4())

            if method == "token_count":
                token_count = method_kwargs["num_tokens"]
                tokens = encoder.encode(content)

                for chunk_num, i in enumerate(
                    range(0, len(tokens), token_count), start=1
                ):
                    chunk_tokens = tokens[i : i + token_count]
                    chunk = encoder.decode(chunk_tokens)

                    result = item.copy()
                    result.update(
                        {
                            f"{split_key}_chunk": chunk,
                            f"{self.name}_id": doc_id,
                            f"{self.name}_chunk_num": chunk_num,
                        }
                    )
                    results.append(result)

            elif method == "delimiter":
                delimiter = method_kwargs["delimiter"]
                num_splits_to_group =
method_kwargs.get("num_splits_to_group", 1)
                chunks = content.split(delimiter)

                # Get rid of empty chunks
                chunks = [chunk for chunk in chunks if chunk.strip()]

                for chunk_num, i in enumerate(
                    range(0, len(chunks), num_splits_to_group), start=1
                ):
                    grouped_chunks = chunks[i : i + num_splits_to_group]
                    joined_chunk = delimiter.join(grouped_chunks).strip()

                    result = item.copy()
                    result.update(
                        {
                            f"{split_key}_chunk": joined_chunk,
                            f"{self.name}_id": doc_id,
                            f"{self.name}_chunk_num": chunk_num,
                        }
                    )
```

```
                            results.append(result)

                return results, cost
```

`docetl.operations.gather.GatherOperation`

Bases: `BaseOperation`

A class that implements a gather operation on input data, adding contextual information from surrounding chunks.

This class extends BaseOperation to: 1. Group chunks by their document ID. 2. Order chunks within each group. 3. Add peripheral context to each chunk based on the configuration. 4. Include headers for each chunk and its upward hierarchy. 5. Return results containing the rendered chunks with added context, including information about skipped characters and headers.

Source code in `docetl/operations/gather.py`

```python
class GatherOperation(BaseOperation):
    """
    A class that implements a gather operation on input data, adding contextual information from surrounding chunks.

    This class extends BaseOperation to:
    1. Group chunks by their document ID.
    2. Order chunks within each group.
    3. Add peripheral context to each chunk based on the configuration.
    4. Include headers for each chunk and its upward hierarchy.
    5. Return results containing the rendered chunks with added context, including information about skipped characters and headers.
    """

    class schema(BaseOperation.schema):
        type: str = "gather"
        content_key: str
        doc_id_key: str
        order_key: str
        peripheral_chunks: dict[str, Any] | None = None
        doc_header_key: str | None = None
        main_chunk_start: str | None = None
        main_chunk_end: str | None = None

        @field_validator("peripheral_chunks")
        def validate_peripheral_chunks(cls, v):
            for direction in ["previous", "next"]:
                if direction not in v:
                    continue
                for section in ["head", "middle", "tail"]:
                    if section in v[direction]:
                        section_config = v[direction][section]
                        if section != "middle" and "count" not in section_config:
                            raise ValueError(
                                f"Missing 'count' in {direction}.{section} configuration"
                            )
            return v

    def __init__(self, *args: Any, **kwargs: Any) -> None:
        """
        Initialize the GatherOperation.

        Args:
            *args: Variable length argument list.
            **kwargs: Arbitrary keyword arguments.
        """
        super().__init__(*args, **kwargs)

    def syntax_check(self) -> None:
        """Perform a syntax check on the operation configuration."""
        # Validate the schema using Pydantic
        self.schema(**self.config)

    def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
```

```
        """
        Execute the gather operation on the input data.

        Args:
            input_data (list[dict]): The input data to process.

        Returns:
            tuple[list[dict], float]: A tuple containing the processed
results and the cost of the operation.
        """
        content_key = self.config["content_key"]
        doc_id_key = self.config["doc_id_key"]
        order_key = self.config["order_key"]
        peripheral_config = self.config.get("peripheral_chunks", {})
        main_chunk_start = self.config.get(
            "main_chunk_start", "--- Begin Main Chunk ---"
        )
        main_chunk_end = self.config.get("main_chunk_end", "--- End Main
Chunk ---")
        doc_header_key = self.config.get("doc_header_key", None)
        results = []
        cost = 0.0

        # Group chunks by document ID
        grouped_chunks = {}
        for item in input_data:
            doc_id = item[doc_id_key]
            if doc_id not in grouped_chunks:
                grouped_chunks[doc_id] = []
            grouped_chunks[doc_id].append(item)

        # Process each group of chunks
        for chunks in grouped_chunks.values():
            # Sort chunks by their order within the document
            chunks.sort(key=lambda x: x[order_key])

            # Process each chunk with its peripheral context and headers
            for i, chunk in enumerate(chunks):
                rendered_chunk = self.render_chunk_with_context(
                    chunks,
                    i,
                    peripheral_config,
                    content_key,
                    order_key,
                    main_chunk_start,
                    main_chunk_end,
                    doc_header_key,
                )

                result = chunk.copy()
                result[f"{content_key}_rendered"] = rendered_chunk
                results.append(result)

        return results, cost

    def render_chunk_with_context(
        self,
        chunks: list[dict],
        current_index: int,
        peripheral_config: dict,
        content_key: str,
```

```
        order_key: str,
        main_chunk_start: str,
        main_chunk_end: str,
        doc_header_key: str,
    ) -> str:
        """
        Render a chunk with its peripheral context and headers.

        Args:
            chunks (list[dict]): List of all chunks in the document.
            current_index (int): Index of the current chunk being
processed.
            peripheral_config (dict): Configuration for peripheral
chunks.
            content_key (str): Key for the content in each chunk.
            order_key (str): Key for the order of each chunk.
            main_chunk_start (str): String to mark the start of the main
chunk.
            main_chunk_end (str): String to mark the end of the main
chunk.
            doc_header_key (str): The key for the headers in the current
chunk.

        Returns:
            str: Renderted chunk with context and headers.
        """

        # If there are no peripheral chunks, return the main chunk
        if not peripheral_config:
            return chunks[current_index][content_key]

        combined_parts = ["--- Previous Context ---"]

        combined_parts.extend(
            self.process_peripheral_chunks(
                chunks[:current_index],
                peripheral_config.get("previous", {}),
                content_key,
                order_key,
            )
        )
        combined_parts.append("--- End Previous Context ---\n")

        # Process main chunk
        main_chunk = chunks[current_index]
        if headers := self.render_hierarchy_headers(
            main_chunk, chunks[: current_index + 1], doc_header_key
        ):
            combined_parts.append(headers)
        combined_parts.extend(
            (
                f"{main_chunk_start}",
                f"{main_chunk[content_key]}",
                f"{main_chunk_end}",
                "\n--- Next Context ---",
            )
        )
        combined_parts.extend(
            self.process_peripheral_chunks(
                chunks[current_index + 1 :],
                peripheral_config.get("next", {}),
```

```
                content_key,
                order_key,
            )
        )
        combined_parts.append("--- End Next Context ---")

        return "\n".join(combined_parts)

    def process_peripheral_chunks(
        self,
        chunks: list[dict],
        config: dict,
        content_key: str,
        order_key: str,
        reverse: bool = False,
    ) -> list[str]:
        """
        Process peripheral chunks according to the configuration.

        Args:
            chunks (list[dict]): List of chunks to process.
            config (dict): Configuration for processing peripheral
chunks.
            content_key (str): Key for the content in each chunk.
            order_key (str): Key for the order of each chunk.
            reverse (bool, optional): Whether to process chunks in
reverse order. Defaults to False.

        Returns:
            list[str]: List of processed chunk strings.
        """
        if reverse:
            chunks = list(reversed(chunks))

        processed_parts = []
        included_chunks = []
        total_chunks = len(chunks)

        head_config = config.get("head", {})
        tail_config = config.get("tail", {})

        head_count = int(head_config.get("count", 0))
        tail_count = int(tail_config.get("count", 0))
        in_skip = False
        skip_char_count = 0

        for i, chunk in enumerate(chunks):
            if i < head_count:
                section = "head"
            elif i >= total_chunks - tail_count:
                section = "tail"
            elif "middle" in config:
                section = "middle"
            else:
                # Show number of characters skipped
                skipped_chars = len(chunk[content_key])
                if not in_skip:
                    skip_char_count = skipped_chars
                    in_skip = True
                else:
                    skip_char_count += skipped_chars
```

```
                continue

            if in_skip:
                processed_parts.append(
                    f"[... {skip_char_count} characters skipped ...]"
                )
                in_skip = False
                skip_char_count = 0

            section_config = config.get(section, {})
            section_content_key = section_config.get("content_key",
content_key)

            is_summary = section_content_key != content_key
            summary_suffix = " (Summary)" if is_summary else ""

            chunk_prefix = f"[Chunk {chunk[order_key]}{summary_suffix}]"
            processed_parts.extend((chunk_prefix, f"
{chunk[section_content_key]}"))
            included_chunks.append(chunk)

        if in_skip:
            processed_parts.append(f"[... {skip_char_count} characters
skipped ...]")

        if reverse:
            processed_parts = list(reversed(processed_parts))

        return processed_parts

    def render_hierarchy_headers(
        self,
        current_chunk: dict,
        chunks: list[dict],
        doc_header_key: str,
    ) -> str:
        """
        Render headers for the current chunk's hierarchy.

        Args:
            current_chunk (dict): The current chunk being processed.
            chunks (list[dict]): List of chunks up to and including the
current chunk.
            doc_header_key (str): The key for the headers in the current
chunk.
        Returns:
            str: Renderted headers in the current chunk's hierarchy.
        """
        current_hierarchy = {}

        if doc_header_key is None:
            return ""

        # Find the largest/highest level in the current chunk
        current_chunk_headers = current_chunk.get(doc_header_key, [])

        # If there are no headers in the current chunk, return an empty
string
        if not current_chunk_headers:
            return ""
```

```
309
310         highest_level = float("inf")  # Initialize with positive infinity
311         for header_info in current_chunk_headers:
312             try:
313                 level = header_info.get("level")
314                 if level is not None and level < highest_level:
315                     highest_level = level
316             except Exception as e:
317                 self.runner.console.log(f"[red]Error processing header:
318 {e}[/red]")
319                 self.runner.console.log(f"[red]Header: {header_info}
320 [/red]")
321                 return ""
322
323         # If no headers found in the current chunk, set highest_level to
324 None
325         if highest_level == float("inf"):
326             highest_level = None
327
            for chunk in chunks:
                for header_info in chunk.get(doc_header_key, []):
                    try:
                        header = header_info["header"]
                        level = header_info["level"]
                        if header and level:
                            current_hierarchy[level] = header
                        # Clear lower levels when a higher level header is
    found
                        for lower_level in range(level + 1,
    len(current_hierarchy) + 1):
                            if lower_level in current_hierarchy:
                                current_hierarchy[lower_level] = None
                    except Exception as e:
                        self.runner.console.log(f"[red]Error processing
    header: {e}[/red]")
                        self.runner.console.log(f"[red]Header: {header_info}
    [/red]")
                        return ""

            rendered_headers = [
                f"{'#' * level} {header}"
                for level, header in sorted(current_hierarchy.items())
                if header is not None and (highest_level is None or level <
    highest_level)
            ]
            rendered_headers = " > ".join(rendered_headers)
            return f"_Current Section:_ {rendered_headers}" if
    rendered_headers else ""
```

`__init__(*args, **kwargs)`

Initialize the GatherOperation.

Parameters:

Name	Type	Description	Default
`*args`	`Any`	Variable length argument list.	`()`
`**kwargs`	`Any`	Arbitrary keyword arguments.	`{}`

Source code in `docetl/operations/gather.py`

```
44  def __init__(self, *args: Any, **kwargs: Any) -> None:
45      """
46      Initialize the GatherOperation.
47
48      Args:
49          *args: Variable length argument list.
50          **kwargs: Arbitrary keyword arguments.
51      """
52      super().__init__(*args, **kwargs)
```

`execute(input_data)`

Execute the gather operation on the input data.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	The input data to process.	*required*

Returns:

Type	Description
`tuple[list[dict], float]`	tuple[list[dict], float]: A tuple containing the processed results and the cost of the operation.

Source code in `docetl/operations/gather.py`

```python
def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
    """
    Execute the gather operation on the input data.

    Args:
        input_data (list[dict]): The input data to process.

    Returns:
        tuple[list[dict], float]: A tuple containing the processed
results and the cost of the operation.
    """
    content_key = self.config["content_key"]
    doc_id_key = self.config["doc_id_key"]
    order_key = self.config["order_key"]
    peripheral_config = self.config.get("peripheral_chunks", {})
    main_chunk_start = self.config.get(
        "main_chunk_start", "--- Begin Main Chunk ---"
    )
    main_chunk_end = self.config.get("main_chunk_end", "--- End Main
Chunk ---")
    doc_header_key = self.config.get("doc_header_key", None)
    results = []
    cost = 0.0

    # Group chunks by document ID
    grouped_chunks = {}
    for item in input_data:
        doc_id = item[doc_id_key]
        if doc_id not in grouped_chunks:
            grouped_chunks[doc_id] = []
        grouped_chunks[doc_id].append(item)

    # Process each group of chunks
    for chunks in grouped_chunks.values():
        # Sort chunks by their order within the document
        chunks.sort(key=lambda x: x[order_key])

        # Process each chunk with its peripheral context and headers
        for i, chunk in enumerate(chunks):
            rendered_chunk = self.render_chunk_with_context(
                chunks,
                i,
                peripheral_config,
                content_key,
                order_key,
                main_chunk_start,
                main_chunk_end,
                doc_header_key,
            )

            result = chunk.copy()
            result[f"{content_key}_rendered"] = rendered_chunk
            results.append(result)

    return results, cost
```

`process_peripheral_chunks(chunks, config, content_key, order_key, reverse=False)`

Process peripheral chunks according to the configuration.

Parameters:

Name	Type	Description	Default
`chunks`	`list[dict]`	List of chunks to process.	*required*
`config`	`dict`	Configuration for processing peripheral chunks.	*required*
`content_key`	`str`	Key for the content in each chunk.	*required*
`order_key`	`str`	Key for the order of each chunk.	*required*
`reverse`	`bool`	Whether to process chunks in reverse order. Defaults to False.	`False`

Returns:

Type	Description
`list[str]`	list[str]: List of processed chunk strings.

Source code in `docetl/operations/gather.py`

```python
def process_peripheral_chunks(
    self,
    chunks: list[dict],
    config: dict,
    content_key: str,
    order_key: str,
    reverse: bool = False,
) -> list[str]:
    """
    Process peripheral chunks according to the configuration.

    Args:
        chunks (list[dict]): List of chunks to process.
        config (dict): Configuration for processing peripheral chunks.
        content_key (str): Key for the content in each chunk.
        order_key (str): Key for the order of each chunk.
        reverse (bool, optional): Whether to process chunks in reverse
order. Defaults to False.

    Returns:
        list[str]: List of processed chunk strings.
    """
    if reverse:
        chunks = list(reversed(chunks))

    processed_parts = []
    included_chunks = []
    total_chunks = len(chunks)

    head_config = config.get("head", {})
    tail_config = config.get("tail", {})

    head_count = int(head_config.get("count", 0))
    tail_count = int(tail_config.get("count", 0))
    in_skip = False
    skip_char_count = 0

    for i, chunk in enumerate(chunks):
        if i < head_count:
            section = "head"
        elif i >= total_chunks - tail_count:
            section = "tail"
        elif "middle" in config:
            section = "middle"
        else:
            # Show number of characters skipped
            skipped_chars = len(chunk[content_key])
            if not in_skip:
                skip_char_count = skipped_chars
                in_skip = True
            else:
                skip_char_count += skipped_chars

            continue

        if in_skip:
            processed_parts.append(
```

```
                f"[... {skip_char_count} characters skipped ...]"
            )
            in_skip = False
            skip_char_count = 0

        section_config = config.get(section, {})
        section_content_key = section_config.get("content_key",
content_key)

        is_summary = section_content_key != content_key
        summary_suffix = " (Summary)" if is_summary else ""

        chunk_prefix = f"[Chunk {chunk[order_key]}{summary_suffix}]"
        processed_parts.extend((chunk_prefix, f"
{chunk[section_content_key]}"))
        included_chunks.append(chunk)

    if in_skip:
        processed_parts.append(f"[... {skip_char_count} characters
skipped ...]")

    if reverse:
        processed_parts = list(reversed(processed_parts))

    return processed_parts
```

`render_chunk_with_context(chunks, current_index, peripheral_config, content_key, order_key, main_chunk_start, main_chunk_end, doc_header_key)`

Render a chunk with its peripheral context and headers.

Parameters:

Name	Type	Description	Default
`chunks`	`list[dict]`	List of all chunks in the document.	*required*
`current_index`	`int`	Index of the current chunk being processed.	*required*
`peripheral_config`	`dict`	Configuration for peripheral chunks.	*required*
`content_key`	`str`	Key for the content in each chunk.	*required*
`order_key`	`str`	Key for the order of each chunk.	*required*

Name	Type	Description	Default
`main_chunk_start`	`str`	String to mark the start of the main chunk.	*required*
`main_chunk_end`	`str`	String to mark the end of the main chunk.	*required*
`doc_header_key`	`str`	The key for the headers in the current chunk.	*required*

Returns:

Name	Type	Description
`str`	`str`	Renderted chunk with context and headers.

Source code in `docetl/operations/gather.py`

```python
def render_chunk_with_context(
    self,
    chunks: list[dict],
    current_index: int,
    peripheral_config: dict,
    content_key: str,
    order_key: str,
    main_chunk_start: str,
    main_chunk_end: str,
    doc_header_key: str,
) -> str:
    """
    Render a chunk with its peripheral context and headers.

    Args:
        chunks (list[dict]): List of all chunks in the document.
        current_index (int): Index of the current chunk being processed.
        peripheral_config (dict): Configuration for peripheral chunks.
        content_key (str): Key for the content in each chunk.
        order_key (str): Key for the order of each chunk.
        main_chunk_start (str): String to mark the start of the main
chunk.
        main_chunk_end (str): String to mark the end of the main chunk.
        doc_header_key (str): The key for the headers in the current
chunk.

    Returns:
        str: Renderted chunk with context and headers.
    """

    # If there are no peripheral chunks, return the main chunk
    if not peripheral_config:
        return chunks[current_index][content_key]

    combined_parts = ["--- Previous Context ---"]

    combined_parts.extend(
        self.process_peripheral_chunks(
            chunks[:current_index],
            peripheral_config.get("previous", {}),
            content_key,
            order_key,
        )
    )
    combined_parts.append("--- End Previous Context ---\n")

    # Process main chunk
    main_chunk = chunks[current_index]
    if headers := self.render_hierarchy_headers(
        main_chunk, chunks[: current_index + 1], doc_header_key
    ):
        combined_parts.append(headers)
    combined_parts.extend(
        (
            f"{main_chunk_start}",
            f"{main_chunk[content_key]}",
            f"{main_chunk_end}",
```

```
            "\n--- Next Context ---",
        )
    )
    combined_parts.extend(
        self.process_peripheral_chunks(
            chunks[current_index + 1 :],
            peripheral_config.get("next", {}),
            content_key,
            order_key,
        )
    )
    combined_parts.append("--- End Next Context ---")

    return "\n".join(combined_parts)
```

`render_hierarchy_headers(current_chunk, chunks, doc_header_key)`

Render headers for the current chunk's hierarchy.

Parameters:

Name	Type	Description	Default
`current_chunk`	`dict`	The current chunk being processed.	*required*
`chunks`	`list[dict]`	List of chunks up to and including the current chunk.	*required*
`doc_header_key`	`str`	The key for the headers in the current chunk.	*required*

Returns: str: Renderted headers in the current chunk's hierarchy.

Source code in `docetl/operations/gather.py`

```
def render_hierarchy_headers(
    self,
    current_chunk: dict,
    chunks: list[dict],
    doc_header_key: str,
) -> str:
    """
    Render headers for the current chunk's hierarchy.

    Args:
        current_chunk (dict): The current chunk being processed.
        chunks (list[dict]): List of chunks up to and including the
current chunk.
        doc_header_key (str): The key for the headers in the current
chunk.
    Returns:
        str: Renderted headers in the current chunk's hierarchy.
    """
    current_hierarchy = {}

    if doc_header_key is None:
        return ""

    # Find the largest/highest level in the current chunk
    current_chunk_headers = current_chunk.get(doc_header_key, [])

    # If there are no headers in the current chunk, return an empty
string
    if not current_chunk_headers:
        return ""

    highest_level = float("inf")  # Initialize with positive infinity
    for header_info in current_chunk_headers:
        try:
            level = header_info.get("level")
            if level is not None and level < highest_level:
                highest_level = level
        except Exception as e:
            self.runner.console.log(f"[red]Error processing header: {e}
[/red]")
            self.runner.console.log(f"[red]Header: {header_info}[/red]")
            return ""

    # If no headers found in the current chunk, set highest_level to None
    if highest_level == float("inf"):
        highest_level = None

    for chunk in chunks:
        for header_info in chunk.get(doc_header_key, []):
            try:
                header = header_info["header"]
                level = header_info["level"]
                if header and level:
                    current_hierarchy[level] = header
                # Clear lower levels when a higher level header is found
                for lower_level in range(level + 1,
len(current_hierarchy) + 1):
```

```
                    if lower_level in current_hierarchy:
                        current_hierarchy[lower_level] = None
            except Exception as e:
                self.runner.console.log(f"[red]Error processing header:
{e}[/red]")
                self.runner.console.log(f"[red]Header: {header_info}
[/red]")
                return ""

        rendered_headers = [
            f"{'#' * level} {header}"
            for level, header in sorted(current_hierarchy.items())
            if header is not None and (highest_level is None or level <
highest_level)
        ]
        rendered_headers = " > ".join(rendered_headers)
        return f"_Current Section:_ {rendered_headers}" if rendered_headers
else ""
```

`syntax_check()`

Perform a syntax check on the operation configuration.

Source code in `docetl/operations/gather.py`

```
def syntax_check(self) -> None:
    """Perform a syntax check on the operation configuration."""
    # Validate the schema using Pydantic
    self.schema(**self.config)
```

`docetl.operations.unnest.UnnestOperation`

Bases: `BaseOperation`

A class that represents an operation to unnest a list-like or dictionary value in a dictionary into multiple dictionaries.

This operation takes a list of dictionaries and a specified key, and creates new dictionaries based on the value type: - For list-like values: Creates a new dictionary for each element in the list, copying all other key-value pairs. - For dictionary values: Expands specified fields from the nested dictionary into the parent dictionary.

Inherits from

BaseOperation

Usage:

```python
from docetl.operations import UnnestOperation

# Unnesting a list
config_list = {"unnest_key": "tags"}
input_data_list = [
    {"id": 1, "tags": ["a", "b", "c"]},
    {"id": 2, "tags": ["d", "e"]}
]

unnest_op_list = UnnestOperation(config_list)
result_list, _ = unnest_op_list.execute(input_data_list)

# Result will be:
# [
#     {"id": 1, "tags": "a"},
#     {"id": 1, "tags": "b"},
#     {"id": 1, "tags": "c"},
#     {"id": 2, "tags": "d"},
#     {"id": 2, "tags": "e"}
# ]

# Unnesting a dictionary
config_dict = {"unnest_key": "user", "expand_fields": ["name", "age"]}
input_data_dict = [
    {"id": 1, "user": {"name": "Alice", "age": 30, "email":
"alice@example.com"}},
    {"id": 2, "user": {"name": "Bob", "age": 25, "email": "bob@example.com"}}
]

unnest_op_dict = UnnestOperation(config_dict)
result_dict, _ = unnest_op_dict.execute(input_data_dict)

# Result will be:
# [
#     {"id": 1, "name": "Alice", "age": 30, "user": {"name": "Alice", "age":
30, "email": "alice@example.com"}},
#     {"id": 2, "name": "Bob", "age": 25, "user": {"name": "Bob", "age": 25,
"email": "bob@example.com"}}
# ]
```

Source code in `docetl/operations/unnest.py`

```python
class UnnestOperation(BaseOperation):
    """
    A class that represents an operation to unnest a list-like or
dictionary value in a dictionary into multiple dictionaries.

    This operation takes a list of dictionaries and a specified key, and
creates new dictionaries based on the value type:
    - For list-like values: Creates a new dictionary for each element in
the list, copying all other key-value pairs.
    - For dictionary values: Expands specified fields from the nested
dictionary into the parent dictionary.

    Inherits from:
        BaseOperation

    Usage:
    ```python
    from docetl.operations import UnnestOperation

    # Unnesting a list
    config_list = {"unnest_key": "tags"}
    input_data_list = [
        {"id": 1, "tags": ["a", "b", "c"]},
        {"id": 2, "tags": ["d", "e"]}
    ]

    unnest_op_list = UnnestOperation(config_list)
    result_list, _ = unnest_op_list.execute(input_data_list)

    # Result will be:
    # [
    #     {"id": 1, "tags": "a"},
    #     {"id": 1, "tags": "b"},
    #     {"id": 1, "tags": "c"},
    #     {"id": 2, "tags": "d"},
    #     {"id": 2, "tags": "e"}
    # ]

    # Unnesting a dictionary
    config_dict = {"unnest_key": "user", "expand_fields": ["name",
"age"]}
    input_data_dict = [
        {"id": 1, "user": {"name": "Alice", "age": 30, "email":
"alice@example.com"}},
        {"id": 2, "user": {"name": "Bob", "age": 25, "email":
"bob@example.com"}}
    ]

    unnest_op_dict = UnnestOperation(config_dict)
    result_dict, _ = unnest_op_dict.execute(input_data_dict)

    # Result will be:
    # [
    #     {"id": 1, "name": "Alice", "age": 30, "user": {"name": "Alice",
"age": 30, "email": "alice@example.com"}},
    #     {"id": 2, "name": "Bob", "age": 25, "user": {"name": "Bob",
"age": 25, "email": "bob@example.com"}}
```
```

```
    # ]
    ```
    """

    class schema(BaseOperation.schema):
        type: str = "unnest"
        unnest_key: str
        keep_empty: bool | None = None
        expand_fields: list[str] | None = None
        recursive: bool | None = None
        depth: int | None = None

    def execute(self, input_data: list[dict]) -> tuple[list[dict],
float]:
        """
        Executes the unnest operation on the input data.

        Args:
            input_data (list[dict]): A list of dictionaries to process.

        Returns:
            tuple[list[dict], float]: A tuple containing the processed
list of dictionaries
            and a float value (always 0 in this implementation).

        Raises:
            KeyError: If the specified unnest_key is not found in an
input dictionary.
            TypeError: If the value of the unnest_key is not iterable
(list, tuple, set, or dict).
            ValueError: If unnesting a dictionary and 'expand_fields' is
not provided in the config.

        The operation supports unnesting of both list-like values and
dictionary values:

        1. For list-like values (list, tuple, set):
           Each element in the list becomes a separate dictionary in the
output.

        2. For dictionary values:
           The operation expands specified fields from the nested
dictionary into the parent dictionary.
           The 'expand_fields' config parameter must be provided to
specify which fields to expand.

        Examples:
        ```python
        # Unnesting a list
        unnest_op = UnnestOperation({"unnest_key": "colors"})
        input_data = [
            {"id": 1, "colors": ["red", "blue"]},
            {"id": 2, "colors": ["green"]}
        ]
        result, _ = unnest_op.execute(input_data)
        # Result will be:
        # [
        #     {"id": 1, "colors": "red"},
        #     {"id": 1, "colors": "blue"},
        #     {"id": 2, "colors": "green"}
        # ]
```

```
        # Unnesting a dictionary
        unnest_op = UnnestOperation({"unnest_key": "details",
"expand_fields": ["color", "size"]})
        input_data = [
            {"id": 1, "details": {"color": "red", "size": "large",
"stock": 5}},
            {"id": 2, "details": {"color": "blue", "size": "medium",
"stock": 3}}
        ]
        result, _ = unnest_op.execute(input_data)
        # Result will be:
        # [
        #     {"id": 1, "details": {"color": "red", "size": "large",
"stock": 5}, "color": "red", "size": "large"},
        #     {"id": 2, "details": {"color": "blue", "size": "medium",
"stock": 3}, "color": "blue", "size": "medium"}
        # ]
        ```

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

        unnest_key = self.config["unnest_key"]
        recursive = self.config.get("recursive", False)
        depth = self.config.get("depth", None)
        if not depth:
            depth = 1 if not recursive else float("inf")
        results = []

        def unnest_recursive(item, key, level=0):
            if level == 0 and not isinstance(item[key], (list, tuple,
set, dict)):
                raise TypeError(f"Value of unnest key '{key}' is not
iterable")

            if level > 0 and not isinstance(item[key], (list, tuple, set,
dict)):
                return [item]

            if level >= depth:
                return [item]

            if isinstance(item[key], dict):
                expand_fields = self.config.get("expand_fields")
                if expand_fields is None:
                    expand_fields = item[key].keys()
                new_item = copy.deepcopy(item)
                for field in expand_fields:
                    if field in new_item[key]:
                        new_item[field] = new_item[key][field]
                    else:
                        new_item[field] = None
                return [new_item]
            else:
                nested_results = []
                for value in item[key]:
                    new_item = copy.deepcopy(item)
                    new_item[key] = value
```
```

```
                    if recursive and isinstance(value, (list, tuple, set,
dict)):
                        nested_results.extend(
                            unnest_recursive(new_item, key, level + 1)
                        )
                    else:
                        nested_results.append(new_item)
                return nested_results

        for item in input_data:
            if unnest_key not in item:
                raise KeyError(
                    f"Unnest key '{unnest_key}' not found in item. Other
keys are {item.keys()}"
                )

            results.extend(unnest_recursive(item, unnest_key))

            if not item[unnest_key] and self.config.get("keep_empty",
False):
                expand_fields = self.config.get("expand_fields")
                new_item = copy.deepcopy(item)
                if isinstance(item[unnest_key], dict):
                    if expand_fields is None:
                        expand_fields = item[unnest_key].keys()
                    for field in expand_fields:
                        new_item[field] = None
                else:
                    new_item[unnest_key] = None
                results.append(new_item)

        # Assert that no keys are missing after the operation
        if results:
            original_keys = set(input_data[0].keys())
            assert original_keys.issubset(
                set(results[0].keys())
            ), "Keys lost during unnest operation"

        return results, 0
```

`execute(input_data)`

Executes the unnest operation on the input data.

Parameters:

Name	Type	Description	Default
`input_data`	`list[dict]`	A list of dictionaries to process.	*required*

Returns:

Type	Description
`list[dict]`	tuple[list[dict], float]: A tuple containing the processed list of dictionaries
`float`	and a float value (always 0 in this implementation).

Raises:

Type	Description
`KeyError`	If the specified unnest_key is not found in an input dictionary.
`TypeError`	If the value of the unnest_key is not iterable (list, tuple, set, or dict).
`ValueError`	If unnesting a dictionary and 'expand_fields' is not provided in the config.

The operation supports unnesting of both list-like values and dictionary values:

1. For list-like values (list, tuple, set): Each element in the list becomes a separate dictionary in the output.
2. For dictionary values: The operation expands specified fields from the nested dictionary into the parent dictionary. The 'expand_fields' config parameter must be provided to specify which fields to expand.

Examples:

```
# Unnesting a list
unnest_op = UnnestOperation({"unnest_key": "colors"})
input_data = [
    {"id": 1, "colors": ["red", "blue"]},
    {"id": 2, "colors": ["green"]}
]
result, _ = unnest_op.execute(input_data)
# Result will be:
# [
#     {"id": 1, "colors": "red"},
#     {"id": 1, "colors": "blue"},
#     {"id": 2, "colors": "green"}
# ]

# Unnesting a dictionary
unnest_op = UnnestOperation({"unnest_key": "details", "expand_fields":
```

```
["color", "size"]})
input_data = [
    {"id": 1, "details": {"color": "red", "size": "large", "stock": 5}},
    {"id": 2, "details": {"color": "blue", "size": "medium", "stock": 3}}
]
result, _ = unnest_op.execute(input_data)
# Result will be:
# [
#     {"id": 1, "details": {"color": "red", "size": "large", "stock": 5},
"color": "red", "size": "large"},
#     {"id": 2, "details": {"color": "blue", "size": "medium", "stock": 3},
"color": "blue", "size": "medium"}
# ]
```

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

Source code in `docetl/operations/unnest.py`

```python
def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
    """
    Executes the unnest operation on the input data.

    Args:
        input_data (list[dict]): A list of dictionaries to process.

    Returns:
        tuple[list[dict], float]: A tuple containing the processed list
of dictionaries
        and a float value (always 0 in this implementation).

    Raises:
        KeyError: If the specified unnest_key is not found in an input
dictionary.
        TypeError: If the value of the unnest_key is not iterable (list,
tuple, set, or dict).
        ValueError: If unnesting a dictionary and 'expand_fields' is not
provided in the config.

    The operation supports unnesting of both list-like values and
dictionary values:

    1. For list-like values (list, tuple, set):
       Each element in the list becomes a separate dictionary in the
output.

    2. For dictionary values:
       The operation expands specified fields from the nested dictionary
into the parent dictionary.
       The 'expand_fields' config parameter must be provided to specify
which fields to expand.

    Examples:
    ```python
    # Unnesting a list
    unnest_op = UnnestOperation({"unnest_key": "colors"})
    input_data = [
        {"id": 1, "colors": ["red", "blue"]},
        {"id": 2, "colors": ["green"]}
    ]
    result, _ = unnest_op.execute(input_data)
    # Result will be:
    # [
    #     {"id": 1, "colors": "red"},
    #     {"id": 1, "colors": "blue"},
    #     {"id": 2, "colors": "green"}
    # ]

    # Unnesting a dictionary
    unnest_op = UnnestOperation({"unnest_key": "details",
"expand_fields": ["color", "size"]})
    input_data = [
        {"id": 1, "details": {"color": "red", "size": "large", "stock":
5}},
        {"id": 2, "details": {"color": "blue", "size": "medium", "stock":
3}}
```
```

```
    ]
    result, _ = unnest_op.execute(input_data)
    # Result will be:
    # [
    #     {"id": 1, "details": {"color": "red", "size": "large", "stock":
5}, "color": "red", "size": "large"},
    #     {"id": 2, "details": {"color": "blue", "size": "medium",
"stock": 3}, "color": "blue", "size": "medium"}
    # ]
    ```

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

    unnest_key = self.config["unnest_key"]
    recursive = self.config.get("recursive", False)
    depth = self.config.get("depth", None)
    if not depth:
        depth = 1 if not recursive else float("inf")
    results = []

    def unnest_recursive(item, key, level=0):
        if level == 0 and not isinstance(item[key], (list, tuple, set,
dict)):
            raise TypeError(f"Value of unnest key '{key}' is not
iterable")

        if level > 0 and not isinstance(item[key], (list, tuple, set,
dict)):
            return [item]

        if level >= depth:
            return [item]

        if isinstance(item[key], dict):
            expand_fields = self.config.get("expand_fields")
            if expand_fields is None:
                expand_fields = item[key].keys()
            new_item = copy.deepcopy(item)
            for field in expand_fields:
                if field in new_item[key]:
                    new_item[field] = new_item[key][field]
                else:
                    new_item[field] = None
            return [new_item]
        else:
            nested_results = []
            for value in item[key]:
                new_item = copy.deepcopy(item)
                new_item[key] = value
                if recursive and isinstance(value, (list, tuple, set,
dict)):
                    nested_results.extend(
                        unnest_recursive(new_item, key, level + 1)
)
                else:
                    nested_results.append(new_item)
            return nested_results

```
```

```
    for item in input_data:
        if unnest_key not in item:
            raise KeyError(
                f"Unnest key '{unnest_key}' not found in item. Other keys
are {item.keys()}"
            )

        results.extend(unnest_recursive(item, unnest_key))

        if not item[unnest_key] and self.config.get("keep_empty", False):
            expand_fields = self.config.get("expand_fields")
            new_item = copy.deepcopy(item)
            if isinstance(item[unnest_key], dict):
                if expand_fields is None:
                    expand_fields = item[unnest_key].keys()
                for field in expand_fields:
                    new_item[field] = None
            else:
                new_item[unnest_key] = None
            results.append(new_item)

    # Assert that no keys are missing after the operation
    if results:
        original_keys = set(input_data[0].keys())
        assert original_keys.issubset(
            set(results[0].keys())
        ), "Keys lost during unnest operation"

    return results, 0
```