

Pointing to External Data and Custom Parsing

In DocETL, you have full control over your dataset JSONs. These JSONs typically contain objects with key-value pairs, where you can reference external files that you want to process in your pipeline. This referencing mechanism, which we call "pointing", allows DocETL to locate and process external files that require special handling before they can be used in your main pipeline.

Why Use Custom Parsing?

Consider these scenarios where custom parsing of referenced files is beneficial:

- Your dataset JSON references Excel spreadsheets containing sales data.
- You have entries pointing to scanned receipts in PDF format that need OCR processing.
- You want to extract text from Word documents or PowerPoint presentations by referencing their file locations.

In these cases, custom parsing enables you to transform your raw external data into a format that DocETL can process effectively within your pipeline. The pointing mechanism allows DocETL to locate these external files and apply custom parsing seamlessly.

(Pointing in DocETL refers to the practice of including references or paths to external files within your dataset JSON. Instead of embedding the entire content of these files, you simply "point" to their locations, allowing DocETL to access and process them as needed during the pipeline execution.)

Dataset JSON Example

Let's look at a typical dataset JSON file that you might create:

```
[  
  { "id": 1, "excel_path": "sales_data/january_sales.xlsx" },  
  { "id": 2, "excel_path": "sales_data/february_sales.xlsx" }  
]
```

In this example, you've specified paths to Excel files. DocETL will use these paths to locate and process the external files. However, without custom parsing, DocETL wouldn't know how to handle the contents of these files. This is where parsing tools come in handy.

Custom Parsing in Action

1. Configuration

To use custom parsing, you need to define parsing tools in your DocETL configuration file. Here's an example:

```

parsing_tools:
  - name: top_products_report
    function_code: |
      def top_products_report(document: Dict) -> List[Dict]:
          import pandas as pd

          # Read the Excel file
          filename = document["excel_path"]
          df = pd.read_excel(filename)

          # Calculate total sales
          total_sales = df['Sales'].sum()

          # Find top 500 products by sales
          top_products = df.groupby('Product')['Sales'].sum().nlargest(500)

          # Calculate month-over-month growth
          df['Date'] = pd.to_datetime(df['Date'])
          monthly_sales = df.groupby(df['Date'].dt.to_period('M'))
          ['Sales'].sum()
          mom_growth = monthly_sales.pct_change().fillna(0)

          # Prepare the analysis report
          report = [
              f"Total Sales: ${total_sales:,.2f}",
              "\nTop 500 Products by Sales:",
              top_products.to_string(),
              "\nMonth-over-Month Growth:",
              mom_growth.to_string()
          ]

          # Return a list of dicts representing the output
          # The input document will be merged into each output doc,
          # so we can access all original fields from the input doc.
          return [{"sales_analysis": "\n".join(report)}]

datasets:
  sales_reports:
    type: file
    source: local
    path: "sales_data/sales_paths.json"
    parsing:
      - function: top_products_report

  receipts:
    type: file
    source: local
    path: "receipts/receipt_paths.json"

```

```

parsing:
  - input_key: pdf_path
    function: paddleocr_pdf_to_string
    output_key: receipt_text
    ocr_enabled: true
    lang: "en"
  
```

In this configuration:

- We define a custom `top_products_report` function for Excel files.
- We use the built-in `paddleocr_pdf_to_string` parser for PDF files.
- We apply these parsing tools to the external files referenced in the respective datasets.

2. Pipeline Integration

Once you've defined your parsing tools and datasets, you can use the processed data in your pipeline:

```

pipeline:
  steps:
    - name: process_sales
      input: sales_reports
      operations:
        - summarize_sales
    - name: process_receipts
      input: receipts
      operations:
        - extract_receipt_info
  
```

This pipeline will use the parsed data from both Excel files and PDFs for further processing.

How Data Gets Parsed and Formatted

When you run your DocETL pipeline, the parsing tools you've specified in your configuration file are applied to the external files referenced in your dataset JSONs. Here's what happens:

1. DocETL reads your dataset JSON file.
2. For each entry in the dataset, it looks at the parsing configuration you've specified.
3. It applies the appropriate parsing function to the file path provided in the dataset JSON.
4. The parsing function processes the file and returns the data in a format DocETL can work with (typically a list of strings).

Let's look at how this works for our earlier examples:

Excel Files (using top_products_report)

For an Excel file like "sales_data/january_sales.xlsx":

- The top_products_report function reads the Excel file.
- It processes the sales data and generates a report of top-selling products.
- The output might look like this:

```
Top Products Report - January 2023
```

```
1. Widget A - 1500 units sold
2. Gadget B - 1200 units sold
3. Gizmo C - 950 units sold
4. Doohickey D - 800 units sold
5. Thingamajig E - 650 units sold
...
```

```
Total Revenue: $245,000
```

```
Best Selling Category: Electronics
```

PDF Files (using paddleocr_pdf_to_string)

For a PDF file like "receipts/receipt001.pdf":

- The paddleocr_pdf_to_string function reads the PDF file.
- It uses PaddleOCR to perform optical character recognition on each page.
- The function combines the extracted text from all pages into a single string. The output might look like this:

```
RECEIPT
```

```
Store: Example Store
```

```
Date: 2023-05-15
```

```
Items:
```

```
1. Product A - $10.99
2. Product B - $15.50
3. Product C - $7.25
4. Product D - $22.00
```

```
Subtotal: $55.74
```

```
Tax (8%): $4.46
```

```
Total: $60.20
```

```
Payment Method: Credit Card
```

```
Card Number: \*\*\*\* \* \*\*\* \* \* \* \* \* 1234
```

```
Thank you for your purchase!
```

This parsed and formatted data is then passed to the respective operations in your pipeline for further processing.

Running the Pipeline

Once you've set up your pipeline configuration file with the appropriate parsing tools and dataset definitions, you can run your DocETL pipeline. Here's how:

1. Ensure you have DocETL installed in your environment.
2. Open a terminal or command prompt.
3. Navigate to the directory containing your pipeline configuration file.
4. Run the following command:

```
docetl run pipeline.yaml
```

Replace `pipeline.yaml` with the name of your pipeline file if it's different.

When you run this command:

1. DocETL reads your pipeline file.
2. It processes each dataset using the specified parsing tools.
3. The pipeline steps are executed in the order you defined.
4. Any operations you've specified (like `summarize_sales` or `extract_receipt_info`) are applied to the parsed data.
5. The results are saved according to your output configuration.

Built-in Parsing Tools

DocETL provides several built-in parsing tools to handle common file formats and data processing tasks. These tools can be used directly in your configuration by specifying their names in the `function` field of your parsing tools configuration. Here's an overview of the available built-in parsing tools:

Convert an Excel file to a string representation or a list of string representations.

Parameters:

Name	Type	Description	Default
<code>filename</code>	<code>str</code>	Path to the xlsx file.	<code>required</code>
<code>orientation</code>	<code>str</code>	Either "row" or "col" for cell arrangement.	<code>'col'</code>

Name	Type	Description	Default
col_order	list[str] None	List of column names to specify the order.	None
doc_per_sheet	bool	If True, return a list of strings, one per sheet.	False

Returns:

Type	Description
list[str]	list[str]: String representation(s) of the Excel file content.

Source code in `docetl/parsing_tools.py`

```
99  @with_input_output_key
100 def xlsx_to_string(
101     filename: str,
102     orientation: str = "col",
103     col_order: list[str] | None = None,
104     doc_per_sheet: bool = False,
105 ) -> list[str]:
106     """
107         Convert an Excel file to a string representation or a list of string
108         representations.
109
110     Args:
111         filename (str): Path to the xlsx file.
112         orientation (str): Either "row" or "col" for cell arrangement.
113         col_order (list[str] | None): List of column names to specify the
114             order.
115         doc_per_sheet (bool): If True, return a list of strings, one per
116             sheet.
117
118     Returns:
119         list[str]: String representation(s) of the Excel file content.
120     """
121     import openpyxl
122
123     wb = openpyxl.load_workbook(filename)
124
125     def process_sheet(sheet):
126         if col_order:
127             headers = [
128                 col for col in col_order if col in sheet.iter_cols(1,
129 sheet.max_column)
130             ]
131         else:
132             headers = [cell.value for cell in sheet[1]]
133
134         result = []
135         if orientation == "col":
136             for col_idx, header in enumerate(headers, start=1):
137                 column = sheet.cell(1, col_idx).column_letter
138                 column_values = [cell.value for cell in sheet[column]
139 [1:]]
140                 result.append(f"{header}: " + "\n".join(map(str,
141 column_values)))
142                 result.append("") # Empty line between columns
143             else: # row
144                 for row in sheet.iter_rows(min_row=2, values_only=True):
145                     row_dict = {
146                         header: value for header, value in zip(headers, row)
147                     if header
148                     }
149                     result.append(
150                         " | ".join(
151                             [f"{header}: {value}" for header, value in
152                             row_dict.items()]
153                         )
154                     )
155             )
```

```

        return "\n".join(result)

    if doc_per_sheet:
        return [process_sheet(sheet) for sheet in wb.worksheets]
    else:
        return [process_sheet(wb.active)]

```

options: show_root_heading: true heading_level: 3

Read the content of a text file and return it as a list of strings (only one element).

Parameters:

Name	Type	Description	Default
filename	str	Path to the txt or md file.	<i>required</i>

Returns:

Type	Description
list[str]	list[str]: Content of the file as a list of strings.

Source code in `docetl/parsing_tools.py`

```

156     @with_input_output_key
157     def txt_to_string(filename: str) -> list[str]:
158         """
159             Read the content of a text file and return it as a list of strings
160             (only one element).
161
162             Args:
163                 filename (str): Path to the txt or md file.
164
165             Returns:
166                 list[str]: Content of the file as a list of strings.
167             """
168             with open(filename, "r", encoding="utf-8") as file:
169                 return [file.read()]

```

options: show_root_heading: true heading_level: 3

Extract text from a Word document.

Parameters:

Name	Type	Description	Default
filename	str	Path to the docx file.	<i>required</i>

Returns:

Type	Description
list[str]	list[str]: Extracted text from the document.

Source code in `docetl/parsing_tools.py`

```

171     @with_input_output_key
172     def docx_to_string(filename: str) -> list[str]:
173         """
174             Extract text from a Word document.
175
176         Args:
177             filename (str): Path to the docx file.
178
179         Returns:
180             list[str]: Extracted text from the document.
181         """
182         from docx import Document
183
184         doc = Document(filename)
185         return ["\n".join([paragraph.text for paragraph in doc.paragraphs])]
```

options: show_root_heading: true heading_level: 3

Transcribe speech from an audio file to text using Whisper model via litellm. If the file is larger than 25 MB, it's split into 10-minute chunks with 30-second overlap.

Parameters:

Name	Type	Description	Default
filename	str	Path to the mp3 or mp4 file.	<i>required</i>

Returns:

Type	Description
list[str]	list[str]: Transcribed text.

Source code in `docetl/parsing_tools.py`

```

52     @with_input_output_key
53     def whisper_speech_to_text(filename: str) -> list[str]:
54         """
55             Transcribe speech from an audio file to text using Whisper model via
56             litellm.
57             If the file is larger than 25 MB, it's split into 10-minute chunks
58             with 30-second overlap.
59
60             Args:
61                 filename (str): Path to the mp3 or mp4 file.
62
63             Returns:
64                 list[str]: Transcribed text.
65             """
66
67     from litellm import transcription
68
69     file_size = os.path.getsize(filename)
70     if file_size > 25 * 1024 * 1024: # 25 MB in bytes
71         from pydub import AudioSegment
72
73         audio = AudioSegment.from_file(filename)
74         chunk_length = 10 * 60 * 1000 # 10 minutes in milliseconds
75         overlap = 30 * 1000 # 30 seconds in milliseconds
76
77         chunks = []
78         for i in range(0, len(audio), chunk_length - overlap):
79             chunk = audio[i : i + chunk_length]
80             chunks.append(chunk)
81
82         transcriptions = []
83
84         for i, chunk in enumerate(chunks):
85             buffer = io.BytesIO()
86             buffer.name = f"temp_chunk_{i}_{os.path.basename(filename)}"
87             chunk.export(buffer, format="mp3")
88             buffer.seek(0) # Reset buffer position to the beginning
89
90             response = transcription(model="whisper-1", file=buffer)
91             transcriptions.append(response.text)
92
93         return transcriptions
94     else:
95         with open(filename, "rb") as audio_file:
96             response = transcription(model="whisper-1", file=audio_file)
97
98         return [response.text]

```

options: show_root_heading: true heading_level: 3

Extract text from a PowerPoint presentation.

Parameters:

Name	Type	Description	Default
filename	str	Path to the pptx file.	<i>required</i>
doc_per_slide	bool	If True, return each slide as a separate document. If False, return the entire presentation as one document.	False

Returns:

Type	Description
list[str]	list[str]: Extracted text from the presentation. If doc_per_slide is True, each string in the list represents a single slide. Otherwise, the list contains a single string with all slides' content.

Source code in `docetl/parsing_tools.py`

```

188     @with_input_output_key
189     def ppx_to_string(filename: str, doc_per_slide: bool = False) ->
190         list[str]:
191             """
192                 Extract text from a PowerPoint presentation.
193
194             Args:
195                 filename (str): Path to the ppx file.
196                 doc_per_slide (bool): If True, return each slide as a separate
197                     document. If False, return the entire presentation as one
198                     document.
199
200             Returns:
201                 list[str]: Extracted text from the presentation. If doc_per_slide
202                     is True, each string in the list represents a single slide.
203                     Otherwise, the list contains a single string with all slides'
204                     content.
205             """
206             from ppx import Presentation
207
208             prs = Presentation(filename)
209             result = []
210
211             for slide in prs.slides:
212                 slide_content = []
213                 for shape in slide.shapes:
214                     if hasattr(shape, "text"):
215                         slide_content.append(shape.text)
216
217                     if doc_per_slide:
218                         result.append("\n".join(slide_content))
219                     else:
220                         result.extend(slide_content)
221
222                     if not doc_per_slide:
223                         result = ["\n".join(result)]
224
225             return result

```

options: show_root_heading: true heading_level: 3

Note to developers: We used [this documentation](#) as a reference.

This function uses Azure Document Intelligence to extract text from documents. To use this function, you need to set up an Azure Document Intelligence resource:

1. [Create an Azure account](#) if you don't have one
2. Set up a Document Intelligence resource in the [Azure portal](#)
3. Once created, find the resource's endpoint and key in the Azure portal
4. Set these as environment variables:

5. DOCUMENTINTELLIGENCE_API_KEY: Your Azure Document Intelligence API key
6. DOCUMENTINTELLIGENCE_ENDPOINT: Your Azure Document Intelligence endpoint URL

The function will use these credentials to authenticate with the Azure service. If the environment variables are not set, the function will raise a `ValueError`.

The Azure Document Intelligence client is then initialized with these credentials. It sends the document (either as a file or URL) to Azure for analysis. The service processes the document and returns structured information about its content.

This function then extracts the text content from the returned data, applying any specified formatting options (like including line numbers or font styles). The extracted text is returned as a list of strings, with each string representing either a page (if `doc_per_page` is `True`) or the entire document.

Parameters:

Name	Type	Description	Default
<code>filename</code>	<code>str</code>	Path to the file to be analyzed or URL of the document if <code>use_url</code> is <code>True</code> .	<code>required</code>
<code>use_url</code>	<code>bool</code>	If <code>True</code> , treat <code>filename</code> as a URL. Defaults to <code>False</code> .	<code>False</code>
<code>include_line_numbers</code>	<code>bool</code>	If <code>True</code> , include line numbers in the output. Defaults to <code>False</code> .	<code>False</code>
<code>include_handwritten</code>	<code>bool</code>	If <code>True</code> , include handwritten text in the output. Defaults to <code>False</code> .	<code>False</code>
<code>include_font_styles</code>	<code>bool</code>	If <code>True</code> , include font style information in the output. Defaults to <code>False</code> .	<code>False</code>
<code>include_selection_marks</code>	<code>bool</code>	If <code>True</code> , include selection marks in the output.	<code>False</code>

Name	Type	Description	Default	
		Defaults to False.		
doc_per_page	bool	If True, return each page as a separate document. Defaults to False.	False	

Returns:

Type	Description
list[str]	list[str]: Extracted text from the document. If doc_per_page is True, each string in the list represents a single page. Otherwise, the list contains a single string with all pages' content.

Raises:

Type	Description
ValueError	If DOCUMENTINTELLIGENCE_API_KEY or DOCUMENTINTELLIGENCE_ENDPOINT environment variables are not set.

Source code in `docetl/parsing_tools.py`

```
226     @with_input_output_key
227     def azure_di_read(
228         filename: str,
229         use_url: bool = False,
230         include_line_numbers: bool = False,
231         include_handwritten: bool = False,
232         include_font_styles: bool = False,
233         include_selection_marks: bool = False,
234         doc_per_page: bool = False,
235     ) -> list[str]:
236         """
237             > Note to developers: We used [this documentation]
238             (https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/how-to-guides/use-sdk-rest-api?view=doc-intel-4.0.0&tabs=windows&pivots=programming-language-python) as a reference.
239
240             This function uses Azure Document Intelligence to extract text from
241             documents.
242
243             To use this function, you need to set up an Azure Document
244             Intelligence resource:
245
246
247                 1. [Create an Azure account] (https://azure.microsoft.com/) if you
248                     don't have one
249                     2. Set up a Document Intelligence resource in the [Azure portal]
250                         (https://portal.azure.com/#create/Microsoft.CognitiveServicesFormRecognizer)
251                     3. Once created, find the resource's endpoint and key in the Azure
252                         portal
253                     4. Set these as environment variables:
254                         - DOCUMENTINTELLIGENCE_API_KEY: Your Azure Document Intelligence
255                         API key
256                         - DOCUMENTINTELLIGENCE_ENDPOINT: Your Azure Document Intelligence
257                         endpoint URL
258
259
260             The function will use these credentials to authenticate with the
261             Azure service.
262
263             If the environment variables are not set, the function will raise a
264             ValueError.
265
266             The Azure Document Intelligence client is then initialized with these
267             credentials.
268
269             It sends the document (either as a file or URL) to Azure for
270             analysis.
271
272             The service processes the document and returns structured information
273             about its content.
274
275             This function then extracts the text content from the returned data,
276             applying any specified formatting options (like including line
277             numbers or font styles).
278
279             The extracted text is returned as a list of strings, with each string
280             representing either a page (if doc_per_page is True) or the entire
281             document.
282
283             Args:
284                 filename (str): Path to the file to be analyzed or URL of the
285                 document if use_url is True.
286                 use_url (bool, optional): If True, treat filename as a URL.
```

```
283     Defaults to False.
284         include_line_numbers (bool, optional): If True, include line
285         numbers in the output. Defaults to False.
286         include_handwritten (bool, optional): If True, include
287         handwritten text in the output. Defaults to False.
288         include_font_styles (bool, optional): If True, include font style
289         information in the output. Defaults to False.
290         include_selection_marks (bool, optional): If True, include
291         selection marks in the output. Defaults to False.
292         doc_per_page (bool, optional): If True, return each page as a
293         separate document. Defaults to False.
294
295     Returns:
296         list[str]: Extracted text from the document. If doc_per_page is
297         True, each string in the list represents
298             a single page. Otherwise, the list contains a single
299         string with all pages' content.
300
301     Raises:
302         ValueError: If DOCUMENTINTELLIGENCE_API_KEY or
303         DOCUMENTINTELLIGENCE_ENDPOINT environment variables are not set.
304         """
305
306         from azure.ai.documentintelligence import DocumentIntelligenceClient
307         from azure.ai.documentintelligence.models import
308     AnalyzeDocumentRequest
309         from azure.core.credentials import AzureKeyCredential
310
311         key = os.getenv("DOCUMENTINTELLIGENCE_API_KEY")
312         endpoint = os.getenv("DOCUMENTINTELLIGENCE_ENDPOINT")
313
314         if key is None:
315             raise ValueError("DOCUMENTINTELLIGENCE_API_KEY environment
316 variable is not set")
317         if endpoint is None:
318             raise ValueError(
319                 "DOCUMENTINTELLIGENCE_ENDPOINT environment variable is not
320 set"
321         )
322
323         document_analysis_client = DocumentIntelligenceClient(
324             endpoint=endpoint, credential=AzureKeyCredential(key)
325         )
326
327         if use_url:
328             poller = document_analysis_client.begin_analyze_document(
329                 "prebuilt-read", AnalyzeDocumentRequest(url_source=filename)
330             )
331         else:
332             with open(filename, "rb") as f:
333                 poller =
334             document_analysis_client.begin_analyze_document("prebuilt-read", f)
335
336             result = poller.result()
337
338             style_content = []
339             content = []
340
341             if result.styles:
342                 for style in result.styles:
343                     if style.is_handwritten and include_handwritten:
```

```

344         handwritten_text = ",".join(
345             [
346                 result.content[span.offset : span.offset +
347                     span.length]
348                     for span in style.spans
349             ]
350         )
351         style_content.append(f"Handwritten content:
352 {handwritten_text}")
353
354         if style.font_style and include_font_styles:
355             styled_text = ",".join(
356                 [
357                     result.content[span.offset : span.offset +
358                         span.length]
359                     for span in style.spans
360             ]
361         )
362         style_content.append(f"'{style.font_style}' font style:
{styled_text}")
363
364     for page in result.pages:
365         page_content = []
366
367         if page.lines:
368             for line_idx, line in enumerate(page.lines):
369                 if include_line_numbers:
370                     page_content.append(f" Line #{line_idx}:
{line.content}")
371                 else:
372                     page_content.append(f"{line.content}")
373
374         if page.selection_marks and include_selection_marks:
375             # TODO: figure this out
376             for selection_mark_idx, selection_mark in
377                 enumerate(page.selection_marks):
378                 page_content.append(
379                     f"Selection mark #{selection_mark_idx}: State is
'{selection_mark.state}' within bounding polygon "
380                     f"'{selection_mark.polygon}' and has a confidence of
{selection_mark.confidence}"
381                 )
382
383         content.append("\n".join(page_content))
384
385         if doc_per_page:
386             return style_content + content
387         else:
388             return [
389                 "\n\n".join(
390                     [
391                         "\n".join(style_content),
392                         "\n\n".join(
393                             f"Page {i+1}:\n{page_content}"
394                             for i, page_content in enumerate(content)
395                         ),
396                     ],
397                 )
398             ]

```

```
options: heading_level: 3 show_root_heading: true
```

Extract text and image information from a PDF file using PaddleOCR for image-based PDFs.

Note: this is very slow!!

Parameters:

Name	Type	Description	Default
input_path	str	Path to the input PDF file.	required
doc_per_page	bool	If True, return a list of strings, one per page. If False, return a single string.	False
ocr_enabled	bool	Whether to enable OCR for image-based PDFs.	True
lang	str	Language of the PDF file.	'en'

Returns:

Type	Description
list[str]	list[str]: Extracted content as a list of formatted strings.

Source code in `docetl/parsing_tools.py`

```
365     @with_input_output_key
366     def paddleocr_pdf_to_string(
367         input_path: str,
368         doc_per_page: bool = False,
369         ocr_enabled: bool = True,
370         lang: str = "en",
371     ) -> list[str]:
372         """
373             Extract text and image information from a PDF file using PaddleOCR
374             for image-based PDFs.
375
376             **Note: this is very slow!!**
377
378             Args:
379                 input_path (str): Path to the input PDF file.
380                 doc_per_page (bool): If True, return a list of strings, one per
381             page.
382                     If False, return a single string.
383                 ocr_enabled (bool): Whether to enable OCR for image-based PDFs.
384                 lang (str): Language of the PDF file.
385
386             Returns:
387                 list[str]: Extracted content as a list of formatted strings.
388             """
389             import fitz
390             import numpy as np
391             from paddleocr import PaddleOCR
392
393             ocr = PaddleOCR(use_angle_cls=True, lang=lang)
394
395             pdf_content = []
396
397             with fitz.open(input_path) as pdf:
398                 for page_num in range(len(pdf)):
399                     page = pdf[page_num]
400                     text = page.get_text()
401                     images = []
402
403                     # Extract image information
404                     for img_index, img in enumerate(page.get_images(full=True)):
405                         rect = page.get_image_bbox(img)
406                         images.append(f"Image {img_index + 1}: bbox {rect}")
407
408                     page_content = f"Page {page_num + 1}:\n"
409                     page_content += f"Text:{text}\n"
410                     page_content += f"Images:{images}\n"
411
412                     if not text and ocr_enabled:
413                         mat = fitz.Matrix(2, 2)
414                         pix = page.get_pixmap(matrix=mat)
415                         img = np.frombuffer(pix.samples, dtype=np.uint8).reshape(
416                             pix.height, pix.width, 3
417                         )
418
419                         ocr_result = ocr.ocr(img, cls=True)
420                         page_content += f"OCR Results:\n"
421                         for line in ocr_result[0]:
```

```

422             bbox, (text, _) = line
423             page_content += f"{bbox}, {text}\n"
424
425         pdf_content.append(page_content)
426
427     if not doc_per_page:
428         return ["\n\n".join(pdf_content)]
429
430     return pdf_content

```

options: heading_level: 3 show_root_heading: true

Using Function Arguments with Parsing Tools

When using parsing tools in your DocETL configuration, you can pass additional arguments to the parsing functions.

For example, when using the `xlsx_to_string` parsing tool, you can specify options like the orientation of the data, the order of columns, or whether to process each sheet separately. Here's an example of how to use such kwargs in your configuration:

```

datasets:
my_sales:
    type: file
    source: local
    path: "sales_data/sales_paths.json"
    parsing_tools:
        - name: excel_parser
          function: xlsx_to_string
          orientation: row
          col_order: ["Date", "Product", "Quantity", "Price"]
          doc_per_sheet: true

```

Contributing Built-in Parsing Tools

While DocETL provides several built-in parsing tools, the community can always benefit from additional utilities. If you've developed a parsing tool that you think could be useful for others, consider contributing it to the DocETL repository. Here's how you can add new built-in parsing utilities:

1. Fork the DocETL repository on GitHub.
2. Clone your forked repository to your local machine.
3. Navigate to the `docetl/parsing_tools.py` file.
4. Add your new parsing function to this file. The function should also be added to the `PARSING_TOOLS` dictionary.
5. Update the documentation in the function's docstring.

6. Create a pull request to merge your changes into the main DocETL repository.



Guidelines for Contributing Parsing Tools

When contributing a new parsing tool, make sure it follows these guidelines:

- The function should have a clear, descriptive name.
- Include comprehensive docstrings explaining the function's purpose, parameters, and return value. The return value should be a list of strings.
- Handle potential errors gracefully and provide informative error messages.
- If your parser requires additional dependencies, make sure to mention them in the pull request.

Creating Custom Parsing Tools

If the built-in tools don't meet your needs, you can create your own custom parsing tools. Here's how:

1. Define your parsing function in the `parsing_tools` section of your configuration.
2. Ensure your function takes a item (dict) as input and returns a list of items (dicts).
3. Use your custom parser in the `parsing` section of your dataset configuration.

For example:

```
parsing_tools:  
  - name: my_custom_parser  
    function_code: |  
      def my_custom_parser(item: Dict) -> List[Dict]:  
        # Your custom parsing logic here  
        return [processed_data]  
  
datasets:  
  my_dataset:  
    type: file  
    source: local  
    path: "data/paths.json"  
    parsing:  
      - function: my_custom_parser
```

Rate Limiting

When using DocETL, you might have rate limits based on your usage tier with various API providers. To help manage these limits and prevent exceeding them, DocETL allows you to configure rate limits in your YAML configuration file.

Configuring Rate Limits

You can add rate limits to your YAML config by including a `rate_limits` key with specific configurations for different types of API calls. Here's an example of how to set up rate limits:

```
rate_limits:
  embedding_call:
    - count: 1000
      per: 1
      unit: second
  llm_call:
    - count: 1
      per: 1
      unit: second
    - count: 10
      per: 5
      unit: hour
  llm_tokens:
    - count: 1000000
      per: 1
      unit: minute
```

Your YAML configuration should have a `rate_limits` key with the config as shown above. This example sets limits for embedding calls and language model (LLM) calls, with multiple rules for LLM calls to accommodate different time scales.

You can also use rate limits in the Python API, passing in a `rate_limits` dictionary when you initialize the `Pipeline` object.

docetl Core

`docetl.DSLRunner`

Bases: `ConfigWrapper`

DSLRunner orchestrates pipeline execution by building and traversing a DAG of OpContainers. The runner uses a two-phase approach:

1. Build Phase:
 2. Parses YAML config into a DAG of OpContainers
 3. Each operation becomes a node connected to its dependencies
 4. Special handling for equijoins which have two parent nodes
 5. Validates operation syntax and schema compatibility
6. Execution Phase:
 7. Starts from the final operation and pulls data through the DAG
 8. Handles caching/checkpointing of intermediate results
 9. Tracks costs and execution metrics
10. Manages dataset loading and result persistence

The separation between build and execution phases allows for:
- Pipeline validation before any execution
- Cost estimation and optimization
- Partial pipeline execution for testing

Source code in `docetl/runner.py`

```
55  class DSLRunner(ConfigWrapper):
56      """
57      DSLRunner orchestrates pipeline execution by building and traversing
58      a DAG of OpContainers.
59      The runner uses a two-phase approach:
60
61      1. Build Phase:
62          - Parses YAML config into a DAG of OpContainers
63          - Each operation becomes a node connected to its dependencies
64          - Special handling for equijoins which have two parent nodes
65          - Validates operation syntax and schema compatibility
66
67      2. Execution Phase:
68          - Starts from the final operation and pulls data through the DAG
69          - Handles caching/checkpointing of intermediate results
70          - Tracks costs and execution metrics
71          - Manages dataset loading and result persistence
72
73      The separation between build and execution phases allows for:
74      - Pipeline validation before any execution
75      - Cost estimation and optimization
76      - Partial pipeline execution for testing
77      """
78
79  @classproperty
80  def schema(cls):
81      # Accessing the schema loads all operations, so only do this
82      # when we actually need it...
83      # Yes, this means DSLRunner.schema isn't really accessible to
84      # static type checkers. But it /is/ available for dynamic
85      # checking, and for generating json schema.
86
87      OpType = functools.reduce(
88          lambda a, b: a | b, [op.schema for op in
89      get_operations().values()])
90      )
91      # More pythonic implementation of the above, but only works in
92      # python 3.11:
93      # OpType = Union[*[op.schema for op in
94      get_operations().values()]]
95
96      class Pipeline(BaseModel):
97          config: dict[str, Any] | None
98          parsing_tools: list[schemas.ParsingTool] | None
99          datasets: dict[str, schemas.Dataset]
100         operations: list[OpType]
101         pipeline: schemas.PipelineSpec
102
103     return Pipeline
104
105  @classproperty
106  def json_schema(cls):
107      return cls.schema.model_json_schema()
108
109  def __init__(self, config: dict, max_threads: int | None = None,
110 **kwargs):
111      """
```

```

112     Initialize the DSLRunner with a YAML configuration file.
113
114     Args:
115         max_threads (int, optional): Maximum number of threads to
116         use. Defaults to None.
117         """
118     super().__init__(
119         config,
120         base_name=kwargs.pop("base_name", None),
121         yaml_file_suffix=kwargs.pop("yaml_file_suffix", None),
122         max_threads=max_threads,
123         **kwargs,
124     )
125     self.total_cost = 0
126     self._initialize_state()
127     self._setup_parsing_tools()
128     self._build_operation_graph(config)
129     self._compute_operation_hashes()
130
131     # Run initial validation
132     self._from_df_accessors = kwargs.get("from_df_accessors", False)
133     if not self._from_df_accessors:
134         self.syntax_check()
135
136     def _initialize_state(self) -> None:
137         """Initialize basic runner state and datasets"""
138         self.datasets = {}
139         self.intermediate_dir = (
140             self.config.get("pipeline", {}).get("output",
141             {}).get("intermediate_dir")
142         )
143
144     def _setup_parsing_tools(self) -> None:
145         """Set up parsing tools from configuration"""
146         self.parsing_tool_map = create_parsing_tool_map(
147             self.config.get("parsing_tools", None)
148         )
149
150     def _build_operation_graph(self, config: dict) -> None:
151         """Build the DAG of operations from configuration"""
152         self.config = config
153         self.op_container_map = {}
154         self.last_op_container = None
155
156         for step in self.config["pipeline"]["steps"]:
157             self._validate_step(step)
158
159             if step.get("input"):
160                 self._add_scan_operation(step)
161             else:
162                 self._add_equijoin_operation(step)
163
164             self._add_step_operations(step)
165             self._add_step_boundary(step)
166
167     def _validate_step(self, step: dict) -> None:
168         """Validate step configuration"""
169         assert "name" in step.keys(), f"Step {step} does not have a name"
170         assert "operations" in step.keys(), f"Step {step} does not have
171         `operations`"
172

```

```

173     def _add_scan_operation(self, step: dict) -> None:
174         """Add a scan operation for input datasets"""
175         scan_op_container = OpContainer(
176             f"{step['name']}/scan_{step['input']}",
177             self,
178             {
179                 "type": "scan",
180                 "dataset_name": step["input"],
181                 "name": f"scan_{step['input']}",
182             },
183         )
184         self.op_container_map[f"{step['name']}/scan_{step['input']}"] = (
185             scan_op_container
186         )
187         if self.last_op_container:
188             scan_op_container.add_child(self.last_op_container)
189         self.last_op_container = scan_op_container
190
191     def _add_equijoin_operation(self, step: dict) -> None:
192         """Add an equijoin operation with its scan operations"""
193         equijoin_operation_name = list(step["operations"][0].keys())[0]
194         left_dataset_name = list(step["operations"][0].values())[0]
195         ["left"]
196         right_dataset_name = list(step["operations"][0].values())[0]
197         ["right"]
198
199         left_scan_op_container = OpContainer(
200             f"{step['name']}/scan_{left_dataset_name}",
201             self,
202             {
203                 "type": "scan",
204                 "dataset_name": left_dataset_name,
205                 "name": f"scan_{left_dataset_name}",
206             },
207         )
208         if self.last_op_container:
209             left_scan_op_container.add_child(self.last_op_container)
210         right_scan_op_container = OpContainer(
211             f"{step['name']}/scan_{right_dataset_name}",
212             self,
213             {
214                 "type": "scan",
215                 "dataset_name": right_dataset_name,
216                 "name": f"scan_{right_dataset_name}",
217             },
218         )
219         if self.last_op_container:
220             right_scan_op_container.add_child(self.last_op_container)
221         equijoin_op_container = OpContainer(
222             f"{step['name']}/{equijoin_operation_name}",
223             self,
224             self.find_operation(equijoin_operation_name),
225             left_name=left_dataset_name,
226             right_name=right_dataset_name,
227         )
228
229         equijoin_op_container.add_child(left_scan_op_container)
230         equijoin_op_container.add_child(right_scan_op_container)
231
232         self.last_op_container = equijoin_op_container
233         self.op_container_map[f"

```

```

234     {step['name']}/{equijoin_operation_name}"] = (
235         equijoin_op_container
236     )
237     self.op_container_map[f"{step['name']}/scan_{left_dataset_name}"]
238 = (
239     left_scan_op_container
240 )
241     self.op_container_map[f"
242 [step['name']]/scan_{right_dataset_name}"] = (
243         right_scan_op_container
244     )
245
246     def _add_step_operations(self, step: dict) -> None:
247         """Add operations for a step"""
248         op_start_idx = 1 if not step.get("input") else 0
249
250         for operation_name in step["operations"][op_start_idx:]:
251             if not isinstance(operation_name, str):
252                 raise ValueError(
253                     f"Operation {operation_name} in step {step['name']} "
254                     "should be a string. "
255                     "If you intend for it to be an equijoin, don't "
256                     "specify an input in the step."
257                 )
258
259             op_container = OpContainer(
260                 f"{step['name']}/{operation_name}",
261                 self,
262                 self.find_operation(operation_name),
263             )
264             op_container.add_child(self.last_op_container)
265             self.last_op_container = op_container
266             self.op_container_map[f"{step['name']}/{operation_name}"] =
267             op_container
268
269     def _add_step_boundary(self, step: dict) -> None:
270         """Add a step boundary node"""
271         step_boundary = StepBoundary(
272             f"{step['name']}/boundary",
273             self,
274             {"type": "step_boundary", "name": f"
275             {step['name']}/boundary"}, )
276             step_boundary.add_child(self.last_op_container)
277             self.op_container_map[f"{step['name']}/boundary"] = step_boundary
278             self.last_op_container = step_boundary
279
280     def _compute_operation_hashes(self) -> None:
281         """Compute hashes for operations to enable caching"""
282         op_map = {op["name"]: op for op in self.config["operations"]}
283         self.step_op_hashes = defaultdict(dict)
284
285         for step in self.config["pipeline"]["steps"]:
286             for idx, op in enumerate(step["operations"]):
287                 op_name = op if isinstance(op, str) else list(op.keys())
288
289 [0]
290
291                 all_ops_until_and_including_current = (
292                     [op_map[prev_op] for prev_op in step["operations"]
293
294 [:idx]] + [op_map[op_name]]]
```

```

295             + [self.config.get("system_prompt", {})]
296         )
297
298         for op in all_ops_until_and_including_current:
299             if "model" not in op:
300                 op["model"] = self.default_model
301
302             all_ops_str =
303             json.dumps(all_ops_until_and_including_current)
304             self.step_op_hashes[step["name"]][op_name] =
305             hashlib.sha256(
306                 all_ops_str.encode()
307             ).hexdigest()
308
309     def get_output_path(self, require=False):
310         output_path = self.config.get("pipeline", {}).get("output",
311         {}).get("path")
312         if output_path:
313             if not (
314                 output_path.lower().endswith(".json")
315                 or output_path.lower().endswith(".csv")
316             ):
317                 raise ValueError(
318                     f"Output path '{output_path}' is not a JSON or CSV
319                     file. Please provide a path ending with '.json' or '.csv'."
320                 )
321             elif require:
322                 raise ValueError(
323                     "No output path specified in the configuration. Please
324                     provide an output path ending with '.json' or '.csv' in the configuration
325                     to use the save() method."
326                 )
327
328         return output_path
329
330     def syntax_check(self):
331         """
332             Perform a syntax check on all operations defined in the
333             configuration.
334         """
335         self.console.log("[yellow]Checking operations...[/yellow]")
336
337         # Just validate that it's a json file if specified
338         self.get_output_path()
339         current = self.last_op_container
340
341     try:
342         # Walk the last op container to check syntax
343         op_containers = []
344         if self.last_op_container:
345             op_containers = [self.last_op_container]
346
347         while op_containers:
348             current = op_containers.pop(0)
349             syntax_result = current.syntax_check()
350             self.console.log(syntax_result, end="")
351             # Add all children to the queue
352             op_containers.extend(current.children)
353     except Exception as e:
354         raise ValueError(
355             f"Syntax check failed for operation '{current.name}':"

```

```

356     {str(e)}"
357     )
358
359         self.console.log("[green]✓ All operations passed syntax
360 check[/green]")
361
362     def print_query_plan(self, show_boundaries=False):
363         """
364             Print a visual representation of the entire query plan using
365             indentation and arrows.
366             Operations are color-coded by step to show the pipeline structure
367             while maintaining
368                 dependencies between steps.
369             """
370
371     if not self.last_op_container:
372         self.console.log("\n[bold]Pipeline Steps:[/bold]")
373         self.console.log(
374             Panel("No operations in pipeline", title="Query Plan",
375 width=100)
376         )
377         self.console.log()
378         return
379
380     def _print_op(
381         op: OpContainer, indent: int = 0, step_colors: dict[str, str]
382     | None = None
383     ) -> str:
384         # Handle boundary operations based on show_boundaries flag
385         if isinstance(op, StepBoundary):
386             if show_boundaries:
387                 output = []
388                 indent_str = " " * indent
389                 step_name = op.name.split("/")[0]
390                 color = step_colors.get(step_name, "white")
391                 output.append(
392                     f"{indent_str}[{color}][bold]{op.name}[/bold]"
393                     "/{color}]"
394                 )
395                 output.append(f"{indent_str}Type: step_boundary")
396                 if op.children:
397                     output.append(f"{indent_str}[yellow]▼[/yellow]")
398                     for child in op.children:
399                         output.append(_print_op(child, indent + 1,
400 step_colors))
401
402             return "\n".join(output)
403         elif op.children:
404             return _print_op(op.children[0], indent, step_colors)
405         return ""
406
407         # Build the string for the current operation with indentation
408         indent_str = " " * indent
409         output = []
410
411         # Color code the operation name based on its step
412         step_name = op.name.split("/")[0]
413         color = step_colors.get(step_name, "white")
414         output.append(f"{indent_str}[{color}][bold]{op.name}[/bold]"
415                     "/{color}]")
416         output.append(f"{indent_str}Type: {op.config['type']}")"
417
418         # Add schema if available

```

```

417         if "output" in op.config and "schema" in op.config["output"]:
418             output.append(f"{indent_str}Output Schema:")
419             for field, field_type in op.config["output"]
420                 ["schema"].items():
421                     escaped_type = escape(str(field_type))
422                     output.append(
423                         f"{indent_str} {field}: [bright_white]"
424                         [escaped_type][/bright_white]"
425                     )
426
427             # Add children
428             if op.children:
429                 if op.is_equipjoin:
430                     output.append(f"{indent_str}[yellow]▼ LEFT[/yellow]")
431                     output.append(_print_op(op.children[0], indent + 1,
432                     step_colors))
433                     output.append(f"{indent_str}[yellow]▼"
434                     RIGHT[/yellow]")
435                     output.append(_print_op(op.children[1], indent + 1,
436                     step_colors))
437                 else:
438                     output.append(f"{indent_str}[yellow]▼[/yellow]")
439                     for child in op.children:
440                         output.append(_print_op(child, indent + 1,
441                     step_colors))
442
443             return "\n".join(output)
444
445     # Get all step boundaries and extract unique step names
446     step_boundaries = [
447         op
448         for name, op in self.op_container_map.items()
449         if isinstance(op, StepBoundary)
450     ]
451     step_boundaries.sort(key=lambda x: x.name)
452
453     # Create a color map for steps - using distinct colors
454     colors = ["cyan", "magenta", "green", "yellow", "blue", "red"]
455     step_names = [b.name.split("/")[0] for b in step_boundaries]
456     step_colors = {
457         name: colors[i % len(colors)] for i, name in
458         enumerate(step_names)
459     }
460
461     # Print the legend
462     self.console.log("\n[bold]Pipeline Steps:[/bold]")
463     for step_name, color in step_colors.items():
464         self.console.log(f"[{color}]■[{color}] {step_name}")
465
466     # Print the full query plan starting from the last step boundary
467     query_plan = _print_op(self.last_op_container,
468     step_colors=step_colors)
469     self.console.log(Panel(query_plan, title="Query Plan",
470     width=100))
471     self.console.log()
472
473     def find_operation(self, op_name: str) -> dict:
474         for operation_config in self.config["operations"]:
475             if operation_config["name"] == op_name:
476                 return operation_config
477             raise ValueError(f"Operation '{op_name}' not found in"

```

```
478     configuration.")
479
480     def load_run_save(self) -> float:
481         """
482             Execute the entire pipeline defined in the configuration.
483         """
484         output_path = self.get_output_path(require=True)
485
486         # Print the query plan
487         self.print_query_plan()
488
489         start_time = time.time()
490
491         if self.last_op_container:
492             self.load()
493             self.console.rule("[bold]Pipeline Execution[/bold]")
494             output, _, _ = self.last_op_container.next()
495             self.save(output)
496
497         execution_time = time.time() - start_time
498
499         # Print execution summary
500         summary = (
501             f"Cost: [green]${self.total_cost:.2f}[/green]\n"
502             f"Time: {execution_time:.2f}s\n"
503             + (
504                 f"Cache: [dim]{self.intermediate_dir}[/dim]\n"
505                 if self.intermediate_dir
506                 else ""
507             )
508             + f"Output: [dim]{output_path}[/dim]"
509         )
510         self.console.log(Panel(summary, title="Execution Summary"))
511
512         return self.total_cost
513
514     def load(self) -> None:
515         """
516             Load all datasets defined in the configuration.
517         """
518         datasets = {}
519         self.console.rule("[bold]Loading Datasets[/bold]")
520
521         for name, dataset_config in self.config["datasets"].items():
522             if dataset_config["type"] == "file":
523                 datasets[name] = Dataset(
524                     self,
525                     "file",
526                     dataset_config["path"],
527                     source="local",
528                     parsing=dataset_config.get("parsing", []),
529                     user_defined_parsing_tool_map=self.parsing_tool_map,
530                 )
531                 self.console.log(
532                     f"[green]✓[/green] Loaded dataset '{name}' from
533 {dataset_config['path']}"
534                 )
535             elif dataset_config["type"] == "memory":
536                 datasets[name] = Dataset(
537                     self,
538                     "memory",
```

```

539         dataset_config["path"],
540         source="local",
541         parsing=dataset_config.get("parsing", []),
542         user_defined_parsing_tool_map=self.parsing_tool_map,
543     )
544     self.console.log(
545         f"[green]✓[/green] Loaded dataset '{name}' from in-
546 memory data"
547     )
548     else:
549         raise ValueError(f"Unsupported dataset type:
550 {dataset_config['type']}"))
551
552     self.datasets = {
553         name: (
554             dataset
555             if isinstance(dataset, Dataset)
556             else Dataset(self, "memory", dataset)
557         )
558         for name, dataset in datasets.items()
559     }
560     self.console.log()
561
562 def save(self, data: list[dict]) -> None:
563     """
564     Save the final output of the pipeline.
565     """
566     self.get_output_path(require=True)
567
568     output_config = self.config["pipeline"]["output"]
569     if output_config["type"] == "file":
570         # Create the directory if it doesn't exist
571         if os.path.dirname(output_config["path"]):
572             os.makedirs(os.path.dirname(output_config["path"]),
573             exist_ok=True)
574         if output_config["path"].lower().endswith(".json"):
575             with open(output_config["path"], "w") as file:
576                 json.dump(data, file, indent=2)
577             else: # CSV
578                 import csv
579
580                 with open(output_config["path"], "w", newline="") as
581 file:
582                     writer = csv.DictWriter(file,
583                     fieldnames=data[0].keys())
584                     limited_data = [
585                         {k: d.get(k, None) for k in data[0].keys()} for d
586                     in data
587                         ]
588                         writer.writeheader()
589                         writer.writerows(limited_data)
590                         self.console.log(
591                             f"[green]✓[/green] Saved to [dim]{output_config['path']}
592 [dim]\n"
593                         )
594                         else:
595                             raise ValueError(
596                                 f"Unsupported output type: {output_config['type']}.
597 Supported types: file"
598                         )
599

```

```

600     def _load_from_checkpoint_if_exists(
601         self, step_name: str, operation_name: str
602     ) -> list[dict] | None:
603         if self.intermediate_dir is None or
604             self.config.get("bypass_cache", False):
605                 return None
606
607         intermediate_config_path = os.path.join(
608             self.intermediate_dir, ".docetl_intermediate_config.json"
609         )
610
611         if not os.path.exists(intermediate_config_path):
612             return None
613
614         # Make sure the step and op name is in the checkpoint config path
615         if (
616             step_name not in self.step_op_hashes
617             or operation_name not in self.step_op_hashes[step_name]
618         ):
619             return None
620
621         # See if the checkpoint config is the same as the current step op
622         hash
623         with open(intermediate_config_path, "r") as f:
624             intermediate_config = json.load(f)
625
626         if (
627             intermediate_config.get(step_name, {}).get(operation_name,
628             "") != self.step_op_hashes[step_name][operation_name]
629         ):
630             return None
631
632         checkpoint_path = os.path.join(
633             self.intermediate_dir, step_name, f"{operation_name}.json"
634         )
635
636         # check if checkpoint exists
637         if os.path.exists(checkpoint_path):
638             if f"{step_name}_{operation_name}" not in self.datasets:
639                 self.datasets[f"{step_name}_{operation_name}"] = Dataset(
640                     self, "file", checkpoint_path, "local"
641                 )
642
643             self.console.log(
644                 f"[green]✓[/green] [italic]Loaded checkpoint for
645 operation '{operation_name}' in step '{step_name}' from {checkpoint_path}
646 [/italic]"
647                 )
648
649             return self.datasets[f"
650 {step_name}_{operation_name}"].load()
651         return None
652
653     def clear_intermediate(self) -> None:
654         """
655             Clear the intermediate directory.
656         """
657
658         # Remove the intermediate directory
659         if self.intermediate_dir:
660             shutil.rmtree(self.intermediate_dir)
661             return

```

```

661         raise ValueError("Intermediate directory not set. Cannot clear
662         intermediate.")
663
664     def _save_checkpoint(
665         self, step_name: str, operation_name: str, data: list[dict]
666     ) -> None:
667         """
668             Save a checkpoint of the current data after an operation.
669
670             This method creates a JSON file containing the current state of
671             the data
672                 after an operation has been executed. The checkpoint is saved in
673             a directory
674                 structure that reflects the step and operation names.
675
676             Args:
677                 step_name (str): The name of the current step in the
678                 pipeline.
679                 operation_name (str): The name of the operation that was just
680                 executed.
681                 data (list[dict]): The current state of the data to be
682                 checkpointered.
683
684             Note:
685                 The checkpoint is saved only if a checkpoint directory has
686                 been specified
687                     when initializing the DSLRunner.
688
689             """
690             checkpoint_path = os.path.join(
691                 self.intermediate_dir, step_name, f"{operation_name}.json"
692             )
693             if os.path.dirname(checkpoint_path):
694                 os.makedirs(os.path.dirname(checkpoint_path), exist_ok=True)
695             with open(checkpoint_path, "w") as f:
696                 json.dump(data, f)
697
698             # Update the intermediate config file with the hash for this
699             step/operation
700             # so that future runs can validate and reuse this checkpoint.
701             if self.intermediate_dir:
702                 intermediate_config_path = os.path.join(
703                     self.intermediate_dir, ".docetl_intermediate_config.json"
704                 )
705
706             # Initialize or load existing intermediate configuration
707             if os.path.exists(intermediate_config_path):
708                 try:
709                     with open(intermediate_config_path, "r") as cfg_file:
710                         intermediate_config: dict[str, dict[str, str]] =
711                         json.load(
712                             cfg_file
713                         )
714                 except json.JSONDecodeError:
715                     # If the file is corrupted, start fresh to avoid
716                     crashes
717                         intermediate_config = {}
718                 else:
719                     intermediate_config = {}
720
721             # Ensure nested dict structure exists

```

```

722         step_dict = intermediate_config.setdefault(step_name, {})
723
724         # Write (or overwrite) the hash for the current operation
725         step_dict[operation_name] = self.step_op_hashes[step_name]
726 [operation_name]
727
728         # Persist the updated configuration
729         with open(intermediate_config_path, "w") as cfg_file:
730             json.dump(intermediate_config, cfg_file, indent=2)
731
732         self.console.log(
733             f"[green]✓ [italic]Intermediate saved for operation
734 '{operation_name}' in step '{step_name}' at {checkpoint_path}[/italic]
735 [/green]"
736             )
737
738     def should_optimize(
739         self, step_name: str, op_name: str, **kwargs
740     ) -> tuple[str, float, list[dict[str, Any]], list[dict[str, Any]]]:
741         self.load()
742
743         # Augment the kwargs with the runner's config if not already
744         provided
745         kwargs["litellm_kwargs"] = self.config.get("optimizer_config",
746 {}).get(
747             "litellm_kwargs", {}
748             )
749         kwargs["rewrite_agent_model"] =
750 self.config.get("optimizer_config", {}).get(
751             "rewrite_agent_model", "gpt-4o"
752             )
753         kwargs["judge_agent_model"] = self.config.get("optimizer_config",
754 {}).get(
755             "judge_agent_model", "gpt-4o-mini"
756             )
757
758         builder = Optimizer(self, **kwargs)
759         self.optimizer = builder
760         result = builder.should_optimize(step_name, op_name)
761         return result
762
763     def optimize(
764         self,
765         save: bool = False,
766         return_pipeline: bool = True,
767         **kwargs,
768     ) -> tuple[dict | "DSLRunner", float]:
769
770         if not self.last_op_container:
771             raise ValueError("No operations in pipeline. Cannot
772 optimize.")
773
774         self.load()
775
776         # Augment the kwargs with the runner's config if not already
777         provided
778         kwargs["litellm_kwargs"] = self.config.get("optimizer_config",
779 {}).get(
780             "litellm_kwargs", {}
781             )
782         kwargs["rewrite_agent_model"] =

```

```

783     self.config.get("optimizer_config", {}).get(
784         "rewrite_agent_model", "gpt-4o"
785     )
786     kwargs["judge_agent_model"] = self.config.get("optimizer_config",
787     {}).get(
788         "judge_agent_model", "gpt-4o-mini"
789     )
790
791     save_path = kwargs.get("save_path", None)
792     # Pop the save_path from kwargs
793     kwargs.pop("save_path", None)
794
795     builder = Optimizer(
796         self,
797         **kwargs,
798     )
799     self.optimizer = builder
800     llm_api_cost = builder.optimize()
801     operations_cost = self.total_cost
802     self.total_cost += llm_api_cost
803
804     # Log the cost of optimization
805     self.console.log(
806         f"[green italic]💡 Total cost: ${self.total_cost:.4f}[/green
italic]"
807     )
808     )
809     self.console.log(
810         f"[green italic] ┌ Operation execution cost:
${operations_cost:.4f}[/green italic]"
811     )
812     self.console.log(
813         f"[green italic] └ Optimization cost: ${llm_api_cost:.4f}
[/green italic]"
814     )
815
816     )
817
818     if save:
819         # If output path is provided, save the optimized config to
that path
820
821         if kwargs.get("save_path"):
822             save_path = kwargs["save_path"]
823             if not os.path.isabs(save_path):
824                 save_path = os.path.join(os.getcwd(), save_path)
825             builder.save_optimized_config(save_path)
826             self.optimized_config_path = save_path
827         else:
828             builder.save_optimized_config(f"
829             {self.base_name}_opt.yaml")
830             self.optimized_config_path = f"{self.base_name}_opt.yaml"
831
832         if return_pipeline:
833             return (
834                 DSLRunner(builder.clean_optimized_config(),
self.max_threads),
835                 self.total_cost,
836             )
837
838         return builder.clean_optimized_config(), self.total_cost
839
840     def _run_operation(
841         self,
842         op_config: dict[str, Any],
843     ):

```

```

        input_data: list[dict[str, Any]] | dict[str, Any],
        return_instance: bool = False,
        is_build: bool = False,
    ) -> list[dict[str, Any]] | tuple[list[dict[str, Any]], BaseOperation, float]:
        """
        Run a single operation based on its configuration.

        This method creates an instance of the appropriate operation class and executes it.
        It also updates the total operation cost.

        Args:
            op_config (dict[str, Any]): The configuration of the operation to run.
            input_data (list[dict[str, Any]]): The input data for the operation.
            return_instance (bool, optional): If True, return the operation instance along with the output data.

        Returns:
            list[dict[str, Any]] | tuple[list[dict[str, Any]], BaseOperation, float]:
                If return_instance is False, returns the output data.
                If return_instance is True, returns a tuple of the output data, the operation instance, and the cost.
        """
        operation_class = get_operation(op_config["type"])

        oc_kwargs = {
            "runner": self,
            "config": op_config,
            "default_model": self.config["default_model"],
            "max_threads": self.max_threads,
            "console": self.console,
            "status": self.status,
        }
        operation_instance = operation_class(**oc_kwargs)
        if op_config["type"] == "equijoin":
            output_data, cost = operation_instance.execute(
                input_data["left_data"], input_data["right_data"]
            )
        elif op_config["type"] == "filter":
            output_data, cost = operation_instance.execute(input_data, is_build)
        else:
            output_data, cost = operation_instance.execute(input_data)

        self.total_cost += cost

        if return_instance:
            return output_data, operation_instance
        else:
            return output_data

    def _flush_partial_results(
        self, operation_name: str, batch_index: int, data: list[dict]
    ) -> None:
        """
        Save partial (batch-level) results from an operation to a directory named

```

```
'<operation_name>_batches' inside the intermediate directory.

Args:
    operation_name (str): The name of the operation, e.g.
    'extract_medications'.
    batch_index (int): Zero-based index of the batch.
    data (list[dict]): Batch results to write to disk.
    """
    if not self.intermediate_dir:
        return

    op_batches_dir = os.path.join(
        self.intermediate_dir, f"{operation_name}_batches"
    )
    os.makedirs(op_batches_dir, exist_ok=True)

    # File name: 'batch_0.json', 'batch_1.json', etc.
    checkpoint_path = os.path.join(op_batches_dir,
        f"batch_{batch_index}.json")

    with open(checkpoint_path, "w") as f:
        json.dump(data, f)

    self.console.log(
        f"[green]✓[/green] [italic]Partial checkpoint saved for
        '{operation_name}', "
        f"batch {batch_index} at '{checkpoint_path}'[/italic]"
    )
)
```

`__init__(config, max_threads=None, **kwargs)`

Initialize the DSLRunner with a YAML configuration file.

Parameters:

Name	Type	Description	Default
<code>max_threads</code>	<code>int</code>	Maximum number of threads to use. Defaults to None.	<code>None</code>

Source code in `docetl/runner.py`

```

105     def __init__(self, config: dict, max_threads: int | None = None,
106                  **kwargs):
107         """
108             Initialize the DSLRunner with a YAML configuration file.
109
110         Args:
111             max_threads (int, optional): Maximum number of threads to use.
112             Defaults to None.
113         """
114         super().__init__(
115             config,
116             base_name=kwargs.pop("base_name", None),
117             yaml_file_suffix=kwargs.pop("yaml_file_suffix", None),
118             max_threads=max_threads,
119             **kwargs,
120         )
121         self.total_cost = 0
122         self._initialize_state()
123         self._setup_parsing_tools()
124         self._build_operation_graph(config)
125         self._compute_operation_hashes()
126
127         # Run initial validation
128         self._from_df_accessors = kwargs.get("from_df_accessors", False)
129         if not self._from_df_accessors:
130             self.syntax_check()

```

`clear_intermediate()`

Clear the intermediate directory.

Source code in `docetl/runner.py`

```

593     def clear_intermediate(self) -> None:
594         """
595             Clear the intermediate directory.
596         """
597             # Remove the intermediate directory
598             if self.intermediate_dir:
599                 shutil.rmtree(self.intermediate_dir)
600
601             raise ValueError("Intermediate directory not set. Cannot clear
602                             intermediate.")

```

`load()`

Load all datasets defined in the configuration.

Source code in `docetl/runner.py`

```

469     def load(self) -> None:
470         """
471             Load all datasets defined in the configuration.
472         """
473         datasets = {}
474         self.console.rule("[bold]Loading Datasets[/bold]")
475
476         for name, dataset_config in self.config["datasets"].items():
477             if dataset_config["type"] == "file":
478                 datasets[name] = Dataset(
479                     self,
480                     "file",
481                     dataset_config["path"],
482                     source="local",
483                     parsing=dataset_config.get("parsing", []),
484                     user_defined_parsing_tool_map=self.parsing_tool_map,
485                 )
486                 self.console.log(
487                     f"[green]✓[/green] Loaded dataset '{name}' from
488 {dataset_config['path']}"
489                 )
490             elif dataset_config["type"] == "memory":
491                 datasets[name] = Dataset(
492                     self,
493                     "memory",
494                     dataset_config["path"],
495                     source="local",
496                     parsing=dataset_config.get("parsing", []),
497                     user_defined_parsing_tool_map=self.parsing_tool_map,
498                 )
499                 self.console.log(
500                     f"[green]✓[/green] Loaded dataset '{name}' from in-memory
501 data"
502                 )
503             else:
504                 raise ValueError(f"Unsupported dataset type:
505 {dataset_config['type']}"))
506
507             self.datasets = {
508                 name: (
509                     dataset
510                     if isinstance(dataset, Dataset)
511                     else Dataset(self, "memory", dataset)
512                 )
513                 for name, dataset in datasets.items()
514             }
515             self.console.log()

```

`load_run_save()`

Execute the entire pipeline defined in the configuration.

Source code in `docetl/runner.py`

```
435     def load_run_save(self) -> float:
436         """
437             Execute the entire pipeline defined in the configuration.
438         """
439         output_path = self.get_output_path(require=True)
440
441         # Print the query plan
442         self.print_query_plan()
443
444         start_time = time.time()
445
446         if self.last_op_container:
447             self.load()
448             self.console.rule("[bold]Pipeline Execution[/bold]")
449             output, _, _ = self.last_op_container.next()
450             self.save(output)
451
452         execution_time = time.time() - start_time
453
454         # Print execution summary
455         summary = (
456             f"Cost: [green]${self.total_cost:.2f}[/green]\n"
457             f"Time: {execution_time:.2f}s\n"
458             +
459             f"Cache: [dim]{self.intermediate_dir}[/dim]\n"
460             if self.intermediate_dir
461             else ""
462         )
463         + f"Output: [dim]{output_path}[/dim]"
464     )
465     self.console.log(Panel(summary, title="Execution Summary"))
466
467     return self.total_cost
```

```
print_query_plan(show_boundaries=False)
```

Print a visual representation of the entire query plan using indentation and arrows.
Operations are color-coded by step to show the pipeline structure while maintaining dependencies between steps.

Source code in docetl/runner.py

```
334     def print_query_plan(self, show_boundaries=False):
335         """
336             Print a visual representation of the entire query plan using
337             indentation and arrows.
338             Operations are color-coded by step to show the pipeline structure
339             while maintaining
340                 dependencies between steps.
341             """
342             if not self.last_op_container:
343                 self.console.log("\n[bold]Pipeline Steps:[/bold]")
344                 self.console.log(
345                     Panel("No operations in pipeline", title="Query Plan",
346 width=100)
347                 )
348                 self.console.log()
349             return
350
351     def _print_op(
352         op: OpContainer, indent: int = 0, step_colors: dict[str, str] |
353 None = None
354     ) -> str:
355         # Handle boundary operations based on show_boundaries flag
356         if isinstance(op, StepBoundary):
357             if show_boundaries:
358                 output = []
359                 indent_str = " " * indent
360                 step_name = op.name.split("/")[-1]
361                 color = step_colors.get(step_name, "white")
362                 output.append(
363                     f"{indent_str}[{color}][bold]{op.name}[/{bold}]"
364                     [/{color}]"
365                 )
366                 output.append(f"{indent_str}Type: step_boundary")
367                 if op.children:
368                     output.append(f"{indent_str}[yellow]▼[/yellow]")
369                     for child in op.children:
370                         output.append(_print_op(child, indent + 1,
371 step_colors))
372             return "\n".join(output)
373         elif op.children:
374             return _print_op(op.children[0], indent, step_colors)
375         return ""
376
377         # Build the string for the current operation with indentation
378         indent_str = " " * indent
379         output = []
380
381         # Color code the operation name based on its step
382         step_name = op.name.split("/")[-1]
383         color = step_colors.get(step_name, "white")
384         output.append(f"{indent_str}[{color}][bold]{op.name}[/{bold}]"
385                     [/{color}]")
386         output.append(f"{indent_str}Type: {op.config['type']}")
387
388         # Add schema if available
389         if "output" in op.config and "schema" in op.config["output"]:
390             output.append(f"{indent_str}Output Schema:")
```

```

391         for field, field_type in op.config["output"]
392     ["schema"].items():
393         escaped_type = escape(str(field_type))
394         output.append(
395             f"\n{indent_str} {field}: [{bright_white}]{escaped_type}"
396             [/{bright_white}]"
397         )
398
399         # Add children
400         if op.children:
401             if op.is_equitjoin:
402                 output.append(f"\n{indent_str}[yellow]▼ LEFT[/{yellow}]")
403                 output.append(_print_op(op.children[0], indent + 1,
404 step_colors))
405                 output.append(f"\n{indent_str}[yellow]▼ RIGHT[/{yellow}]")
406                 output.append(_print_op(op.children[1], indent + 1,
407 step_colors))
408             else:
409                 output.append(f"\n{indent_str}[yellow]▼[/{yellow}]")
410                 for child in op.children:
411                     output.append(_print_op(child, indent + 1,
412 step_colors))
413
414         return "\n".join(output)
415
416     # Get all step boundaries and extract unique step names
417     step_boundaries = [
418         op
419         for name, op in self.op_container_map.items()
420         if isinstance(op, StepBoundary)
421     ]
422     step_boundaries.sort(key=lambda x: x.name)
423
424     # Create a color map for steps - using distinct colors
425     colors = ["cyan", "magenta", "green", "yellow", "blue", "red"]
426     step_names = [b.name.split("/")[0] for b in step_boundaries]
427     step_colors = {
428         name: colors[i % len(colors)] for i, name in
429         enumerate(step_names)
430     }
431
432     # Print the legend
433     self.console.log("\n[bold]Pipeline Steps:[/bold]")
434     for step_name, color in step_colors.items():
435         self.console.log(f"[{color}]■[/{color}] {step_name}")
436
437     # Print the full query plan starting from the last step boundary
438     query_plan = _print_op(self.last_op_container,
439     step_colors=step_colors)
440     self.console.log(Panel(query_plan, title="Query Plan", width=100))
441     self.console.log()

```

`save(data)`

Save the final output of the pipeline.

Source code in `docetl/runner.py`

```

514     def save(self, data: list[dict]) -> None:
515         """
516             Save the final output of the pipeline.
517         """
518         self.get_output_path(require=True)
519
520         output_config = self.config["pipeline"]["output"]
521         if output_config["type"] == "file":
522             # Create the directory if it doesn't exist
523             if os.path.dirname(output_config["path"]):
524                 os.makedirs(os.path.dirname(output_config["path"]),
525                 exist_ok=True)
526             if output_config["path"].lower().endswith(".json"):
527                 with open(output_config["path"], "w") as file:
528                     json.dump(data, file, indent=2)
529             else: # CSV
530                 import csv
531
532                 with open(output_config["path"], "w", newline="") as file:
533                     writer = csv.DictWriter(file, fieldnames=data[0].keys())
534                     limited_data = [
535                         {k: d.get(k, None) for k in data[0].keys()} for d in
536                         data
537                     ]
538                     writer.writeheader()
539                     writer.writerows(limited_data)
540                     self.console.log(
541                         f"[green]✓[/green] Saved to [dim]{output_config['path']}\n"
542                         )
543             else:
544                 raise ValueError(
545                     f"Unsupported output type: {output_config['type']}. Supported
546                     types: file"
547                 )

```

`syntax_check()`

Perform a syntax check on all operations defined in the configuration.

Source code in `docetl/runner.py`

```
305     def syntax_check(self):
306         """
307             Perform a syntax check on all operations defined in the
308             configuration.
309             """
310             self.console.log("[yellow]Checking operations...[/yellow]")
311
312             # Just validate that it's a json file if specified
313             self.get_output_path()
314             current = self.last_op_container
315
316             try:
317                 # Walk the last op container to check syntax
318                 op_containers = []
319                 if self.last_op_container:
320                     op_containers = [self.last_op_container]
321
322                 while op_containers:
323                     current = op_containers.pop(0)
324                     syntax_result = current.syntax_check()
325                     self.console.log(syntax_result, end="")
326                     # Add all children to the queue
327                     op_containers.extend(current.children)
328                 except Exception as e:
329                     raise ValueError(
330                         f"Syntax check failed for operation '{current.name}':
331                         {str(e)}")
332
333
334             self.console.log("[green]✓ All operations passed syntax
check[/green]")



```

docetl.Optimizer

Orchestrates the optimization of a DocETL pipeline by analyzing and potentially rewriting operations marked for optimization. Works with the runner's pull-based execution model to maintain lazy evaluation while improving pipeline efficiency.

Source code in `docetl/optimizer.py`

```
48     class Optimizer:
49         """
50             Orchestrates the optimization of a DocETL pipeline by analyzing and
51             potentially rewriting
52                 operations marked for optimization. Works with the runner's pull-
53                 based execution model
54                     to maintain lazy evaluation while improving pipeline efficiency.
55             """
56
57     def __init__(
58         self,
59         runner: "DSLRunner",
60         rewrite_agent_model: str = "gpt-4o",
61         judge_agent_model: str = "gpt-4o-mini",
62         litellm_kwargs: dict[str, Any] = {},
63         resume: bool = False,
64         timeout: int = 60,
65     ):
66         """
67             Initialize the optimizer with a runner instance and
68             configuration.
69                 Sets up optimization parameters, caching, and cost tracking.
70
71             Args:
72                 yaml_file (str): Path to the YAML configuration file.
73                 model (str): The name of the language model to use. Defaults
74                 to "gpt-4o".
75                 resume (bool): Whether to resume optimization from a previous
76                 run. Defaults to False.
77                 timeout (int): Timeout in seconds for operations. Defaults to
78                 60.
79
80             Attributes:
81                 config (Dict): Stores the loaded configuration from the YAML
82                 file.
83                 console (Console): Rich console for formatted output.
84                 max_threads (int): Maximum number of threads for parallel
85                 processing.
86                 base_name (str): Base name used for file paths.
87                 yaml_file_suffix (str): Suffix for YAML configuration files.
88                 runner (DSLRunner): The DSL runner instance.
89                 status: Status tracking for the runner.
90                 optimized_config (Dict): A copy of the original config to be
91                 optimized.
92                 llm_client (LLMClient): Client for interacting with the
93                 language model.
94                 timeout (int): Timeout for operations in seconds.
95                 resume (bool): Whether to resume from previous optimization.
96                 captured_output (CapturedOutput): Captures output during
97                 optimization.
98                 sample_cache (Dict): Maps operation names to tuples of
99                 (output_data, sample_size).
100                optimized_ops_path (str): Path to store optimized operations.
101                sample_size_map (Dict): Maps operation types to sample sizes.
102
103                The method also calls print_optimizer_config() to display the
104                initial configuration.
```

```

105     """
106     self.config = runner.config
107     self.console = runner.console
108     self.max_threads = runner.max_threads
109
110     self.base_name = runner.base_name
111     self.yaml_file_suffix = runner.yaml_file_suffix
112     self.runner = runner
113     self.status = runner.status
114
115     self.optimized_config = copy.deepcopy(self.config)
116
117     # Get the rate limits from the optimizer config
118     rate_limits = self.config.get("optimizer_config",
119     {}).get("rate_limits", {})
120
121     self.llm_client = LLMClient(
122         runner,
123         rewrite_agent_model,
124         judge_agent_model,
125         rate_limits,
126         **litellm_kwargs,
127     )
128     self.timeout = timeout
129     self.resume = resume
130     self.captured_output = CapturedOutput()
131
132     # Add sample cache for build operations
133     self.sample_cache = {} # Maps operation names to (output_data,
134     sample_size)
135
136     home_dir = os.environ.get("DOCETL_HOME_DIR",
137     os.path.expanduser("~/"))
138     cache_dir = os.path.join(home_dir,
139     f".docetl/cache/{runner.yaml_file_suffix}")
140     os.makedirs(cache_dir, exist_ok=True)
141
142     # Hash the config to create a unique identifier
143     config_hash =
144     hashlib.sha256(str(self.config).encode()).hexdigest()
145     self.optimized_ops_path = f"{cache_dir}/{config_hash}.yaml"
146
147     # Update sample size map
148     self.sample_size_map = SAMPLE_SIZE_MAP
149     if self.config.get("optimizer_config", {}).get("sample_sizes",
150     {}):
151         self.sample_size_map.update(self.config["optimizer_config"]
152         ["sample_sizes"])
153
154     if not self.runner._from_df_accessors:
155         self.print_optimizer_config()
156
157     def print_optimizer_config(self):
158         """
159         Print the current configuration of the optimizer.
160
161         This method uses the Rich console to display a formatted output
162         of the optimizer's
163             configuration. It includes details such as the YAML file path,
164             sample sizes for
165                 different operation types, maximum number of threads, the

```

```

166     language model being used,
167     and the timeout setting.
168
169     The output is color-coded and formatted for easy readability,
170     with a header and
171     separator lines to clearly delineate the configuration
172     information.
173     """
174     self.console.log(
175         Panel.fit(
176             "[bold cyan]Optimizer Configuration[/bold cyan]\n"
177             f"[yellow]Sample Size:{self.sample_size_map}\n"
178             f"[yellow]Max Threads:{self.max_threads}\n"
179             f"[yellow]Rewrite Agent Model:{self.llm_client.rewrite_agent_model}\n"
180             f"[yellow]Judge Agent Model:{self.llm_client.judge_agent_model}\n"
181             f"[yellow]Rate Limits:{self.config.get('optimizer_config', {}).get('rate_limits', {})}\n",
182             title="Optimizer Configuration",
183             )
184         )
185     )
186
187
188     def _insert_empty_resolve_operations(self):
189     """
190         Determines whether to insert resolve operations in the pipeline.
191
192         For each reduce operation in the tree, checks if it has any map
193         operation as a descendant
194         without a resolve operation in between. If found, inserts an
195         empty resolve operation
196         right after the reduce operation.
197
198         The method modifies the operation container tree in-place.
199
200         Returns:
201             None
202         """
203
204         if not self.runner.last_op_container:
205             return
206
207         def find_map_without_resolve(container, visited=None):
208             """Helper to find first map descendant without a resolve
209             operation in between."""
210             if visited is None:
211                 visited = set()
212
213             if container.name in visited:
214                 return None
215             visited.add(container.name)
216
217             if not container.children:
218                 return None
219
220             for child in container.children:
221                 if child.config["type"] == "map":
222                     return child
223                 if child.config["type"] == "resolve":
224                     continue
225                 map_desc = find_map_without_resolve(child, visited)
226                 if map_desc:

```

```

227         return map_desc
228     return None
229
230     # Walk down the operation container tree
231     containers_to_check = [self.runner.last_op_container]
232     while containers_to_check:
233         current = containers_to_check.pop(0)
234
235         # Skip if this is a boundary or has no children
236         if isinstance(current, StepBoundary) or not current.children:
237             containers_to_check.extend(current.children)
238             continue
239
240         # Get the step name from the container's name
241         step_name = current.name.split("/")[-1]
242
243         # Check if current container is a reduce operation
244         if current.config["type"] == "reduce" and current.config.get(
245             "synthesize_resolve", True
246         ):
247             reduce_key = current.config.get("reduce_key", "_all")
248             if isinstance(reduce_key, str):
249                 reduce_key = [reduce_key]
250
251             if "_all" not in reduce_key:
252                 # Find map descendant without resolve
253                 map_desc = find_map_without_resolve(current)
254                 if map_desc:
255                     # Synthesize an empty resolver
256                     self.console.log(
257                         "[yellow]Synthesizing empty resolver"
258                         f" {current.name}[/yellow]"
259                     )
260                     self.console.log(
261                         f"  • [cyan]Reduce operation:[/cyan] [bold]"
262                         f"[{current.name}][/bold]"
263                     )
264                     self.console.log(
265                         f"  • [cyan]Step:[/cyan] [bold]{step_name}"
266                         "[/bold]"
267                     )
268
269             # Create new resolve operation config
270             new_resolve_name = (
271
272                 f"synthesized_resolve_{len(self.config['operations'])}"
273             )
274             new_resolve_config = {
275                 "name": new_resolve_name,
276                 "type": "resolve",
277                 "empty": True,
278                 "optimize": True,
279                 "embedding_model": "text-embedding-3-small",
280                 "resolution_model": self.config.get(
281                     "default_model", "gpt-4o-mini"
282                 ),
283                 "comparison_model": self.config.get(
284                     "default_model", "gpt-4o-mini"
285                 ),
286                 "_intermediates": {
287                     "map_prompt": "
```

```

288     map_desc.config.get("prompt"),
289             "reduce_key": reduce_key,
290         },
291     }
292
293     # Add to operations list
294
295     self.config["operations"].append(new_resolve_config)
296
297     # Create new resolve container
298     new_resolve_container = OpContainer(
299         f"{step_name}/{new_resolve_name}",
300         self.runner,
301         new_resolve_config,
302     )
303
304     # Insert the new container between reduce and its
305     children
306     new_resolve_container.children = current.children
307     for child in new_resolve_container.children:
308         child.parent = new_resolve_container
309     current.children = [new_resolve_container]
310     new_resolve_container.parent = current
311
312     # Add to container map
313     self.runner.op_container_map[
314         f"{step_name}/{new_resolve_name}"
315     ] = new_resolve_container
316
317     # Add children to the queue
318
319     containers_to_check.extend(new_resolve_container.children)
320
321     def _add_map_prompts_to_reduce_operations(self):
322         """
323             Add relevant map prompts to reduce operations based on their
324             reduce keys.
325
326             This method walks the operation container tree to find map
327             operations and their
328                 output schemas, then associates those with reduce operations that
329                 use those keys.
330             When a reduce operation is found, it looks through its
331             descendants to find the
332                 relevant map operations and adds their prompts.
333
334             The method modifies the operation container tree in-place.
335         """
336         if not self.runner.last_op_container:
337             return
338
339         def find_map_prompts_for_keys(container, keys, visited=None):
340             """Helper to find map prompts for given keys in the
341             container's descendants."""
342             if visited is None:
343                 visited = set()
344
345             if container.name in visited:
346                 return []
347             visited.add(container.name)
348

```

```

349         prompts = []
350         if container.config["type"] == "map":
351             output_schema = container.config.get("output",
352             {}).get("schema", {})
353             if any(key in output_schema for key in keys):
354                 prompts.append(container.config.get("prompt", ""))
355
356             for child in container.children:
357                 prompts.extend(find_map_prompts_for_keys(child, keys,
358                 visited))
359
360         return prompts
361
362     # Walk down the operation container tree
363     containers_to_check = [self.runner.last_op_container]
364     while containers_to_check:
365         current = containers_to_check.pop(0)
366
367         # Skip if this is a boundary or has no children
368         if isinstance(current, StepBoundary) or not current.children:
369             containers_to_check.extend(current.children)
370             continue
371
372         # If this is a reduce operation, find relevant map prompts
373         if current.config["type"] == "reduce":
374             reduce_keys = current.config.get("reduce_key", [])
375             if isinstance(reduce_keys, str):
376                 reduce_keys = [reduce_keys]
377
378             # Find map prompts in descendants
379             relevant_prompts = find_map_prompts_for_keys(current,
380             reduce_keys)
381
382             if relevant_prompts:
383                 current.config["_intermediates"] =
384             current.config.get(
385                 "_intermediates", {})
386             )
387             current.config["_intermediates"]["last_map_prompt"] =
388             (
389                 relevant_prompts[-1]
390             )
391
392             # Add children to the queue
393             containers_to_check.extend(current.children)
394
395     def should_optimize(
396         self, step_name: str, op_name: str
397     ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]], float]:
398         """
399             Analyzes whether an operation should be optimized by running it
400             on a sample of input data
401             and evaluating potential optimizations. Returns the optimization
402             suggestion and relevant data.
403             """
404             self.console.rule("[bold cyan]Beginning Pipeline Assessment[/bold
405             cyan]")
406
407             self._insert_empty_resolve_operations()
408
409             node_of_interest = self.runner.op_container_map[f"
```

```

410     f'{step_name}/{op_name}"']
411
412         # Run the node_of_interest's children
413         input_data = []
414         for child in node_of_interest.children:
415             input_data.append(
416                 child.next(
417                     is_build=True,
418
419             sample_size_needed=SAMPLE_SIZE_MAP.get(child.config["type"]),
420                     )[0]
421             )
422
423         # Set the step
424         self.captured_output.set_step(step_name)
425
426         # Determine whether we should optimize the node_of_interest
427         if (
428             node_of_interest.config.get("type") == "map"
429             or node_of_interest.config.get("type") == "filter"
430         ):
431             # Create instance of map optimizer
432             map_optimizer = MapOptimizer(
433                 self.runner,
434                 self.runner._run_operation,
435                 is_filter=node_of_interest.config.get("type") ==
436 "filter",
437                 )
438             should_optimize_output, input_data, output_data = (
439                 map_optimizer.should_optimize(node_of_interest.config,
440                 input_data[0])
441                 )
442             elif node_of_interest.config.get("type") == "reduce":
443                 reduce_optimizer = ReduceOptimizer(
444                     self.runner,
445                     self.runner._run_operation,
446                     )
447                 should_optimize_output, input_data, output_data = (
448                     reduce_optimizer.should_optimize(node_of_interest.config,
449                     input_data[0])
450                     )
451             elif node_of_interest.config.get("type") == "resolve":
452                 resolve_optimizer = JoinOptimizer(
453                     self.runner,
454                     node_of_interest.config,
455                     target_recall=self.config.get("optimizer_config", {})
456                     .get("resolve", {})
457                     .get("target_recall", 0.95),
458                     )
459                     _, should_optimize_output =
460             resolve_optimizer.should_optimize(input_data[0])
461
462             # if should_optimize_output is empty, then we should move to
463             the reduce operation
464             if should_optimize_output == "":
465                 return "", [], [], 0.0
466             else:
467                 return "", [], [], 0.0
468
469             # Return the string and operation cost
470             return (

```

```

471         should_optimize_output,
472         input_data,
473         output_data,
474         self.runner.total_cost + self.llm_client.total_cost,
475     )
476
477     def optimize(self) -> float:
478         """
479             Optimizes the entire pipeline by walking the operation DAG and
480             applying
481                 operation-specific optimizers where marked. Returns the total
482             optimization cost.
483             """
484             self.console.rule("[bold cyan]Beginning Pipeline Rewrites[/bold
485 cyan]")
486
487             # If self.resume is True and there's a checkpoint, load it
488             if self.resume:
489                 if os.path.exists(self.optimized_ops_path):
490                     # Load the yaml and change the runner with it
491                     with open(self.optimized_ops_path, "r") as f:
492                         partial_optimized_config = yaml.safe_load(f)
493                         self.console.log(
494                             "[yellow]Loading partially optimized pipeline
from checkpoint...[/yellow]"
495                         )
496
497             self.runner._build_operation_graph(partial_optimized_config)
498             else:
499                 self.console.log(
500                     "[yellow]No checkpoint found, starting optimization
from scratch...[/yellow]"
501                     )
502
503             else:
504                 self._insert_empty_resolve_operations()
505
506             # Start with the last operation container and visit each child
507             self.runner.last_op_container.optimize()
508
509             flush_cache(self.console)
510
511             # Print the query plan
512             self.console.rule("[bold cyan]Optimized Query Plan[/bold cyan]")
513             self.runner.print_query_plan()
514
515             return self.llm_client.total_cost
516
517
518     def _optimize_equipjoin(
519         self,
520         op_config: dict[str, Any],
521         left_name: str,
522         right_name: str,
523         left_data: list[dict[str, Any]],
524         right_data: list[dict[str, Any]],
525         run_operation: Callable[
526             [dict[str, Any], list[dict[str, Any]]], list[dict[str, Any]]
527         ],
528         ) -> tuple[list[dict[str, Any]], dict[str, list[dict[str, Any]]],
529         str, str]:
530             """
531

```

```

532     Optimizes an equijoin operation by analyzing join conditions and
533     potentially inserting
534         map operations to improve join efficiency. Returns the optimized
535     configuration and updated data.
536     """
537     max_iterations = 2
538     new_left_name = left_name
539     new_right_name = right_name
540     new_steps = []
541     for _ in range(max_iterations):
542         join_optimizer = JoinOptimizer(
543             self.runner,
544             op_config,
545             target_recall=self.runner.config.get("optimizer_config",
546             {}),
547             .get("equijoin", {}),
548             .get("target_recall", 0.95),
549
550             estimated_selectivity=self.runner.config.get("optimizer_config", {})
551             .get("equijoin", {})
552             .get("estimated_selectivity", None),
553             )
554             optimized_config, cost, agent_results =
555             join_optimizer.optimize_equijoin(
556                 left_data, right_data
557                 )
558             self.runner.total_cost += cost
559             # Update the operation config with the optimized values
560             op_config.update(optimized_config)
561
562             if not agent_results.get("optimize_map", False):
563                 break # Exit the loop if no more map optimizations are
564             necessary
565
566             # Update the status to indicate we're optimizing a map
567             operation
568             output_key = agent_results["output_key"]
569             if self.runner.status:
570                 self.runner.status.update(
571                     f"Optimizing map operation for {output_key}"
572                     extraction to help with the equijoin"
573                     )
574             map_prompt = agent_results["map_prompt"]
575             dataset_to_transform =
576                 left_data
577                 if agent_results["dataset_to_transform"] == "left"
578                 else right_data
579             )
580
581             # Create a new step for the map operation
582             map_operation = {
583                 "name": f"synthesized_{output_key}_extraction",
584                 "type": "map",
585                 "prompt": map_prompt,
586                 "model": self.config.get("default_model", "gpt-4o-mini"),
587                 "output": {"schema": {output_key: "string"}},
588                 "optimize": False,
589             }
590
591             # Optimize the map operation
592             if map_operation["optimize"]:

```

```

593         dataset_to_transform_sample = (
594             random.sample(dataset_to_transform,
595             self.sample_size_map.get("map"))
596             if self.config.get("optimizer_config", {}).get(
597                 "random_sample", False
598             )
599             else dataset_to_transform[:self.sample_size_map.get("map")]
600         )
601         optimized_map_operations = self._optimize_map(
602             map_operation, dataset_to_transform_sample
603         )
604     else:
605         optimized_map_operations = [map_operation]
606
607     new_step = {
608         "name": f"synthesized_{output_key}_extraction",
609         "input": (
610             left_name
611             if agent_results["dataset_to_transform"] == "left"
612             else right_name
613         ),
614         "operations": [mo["name"] for mo in
615             optimized_map_operations],
616     }
617     if agent_results["dataset_to_transform"] == "left":
618         new_left_name = new_step["name"]
619     else:
620         new_right_name = new_step["name"]
621
622     new_steps.append((new_step["name"], new_step,
623         optimized_map_operations))
624
625     # Now run the optimized map operation on the entire
626     dataset_to_transform
627     for op in optimized_map_operations:
628         dataset_to_transform = run_operation(op,
629             dataset_to_transform)
630
631     # Update the appropriate dataset for the next iteration
632     if agent_results["dataset_to_transform"] == "left":
633         left_data = dataset_to_transform
634     else:
635         right_data = dataset_to_transform
636
637     if self.runner.status:
638         self.runner.status.update(
639             f"Optimizing equijoin operation with {output_key}"
640             extraction"
641         )
642
643     return op_config, new_steps, new_left_name, new_right_name
644
645     def checkpoint_optimized_ops(self) -> None:
646         """
647             Generates the clean config and saves it to the
648             self.optimized_ops_path
649             This is used to resume optimization from a previous run
650             """
651             clean_config = self.clean_optimized_config()
652             with open(self.optimized_ops_path, "w") as f:

```

```
654         yaml.safe_dump(clean_config, f, default_flow_style=False,
655                     width=80)
656
657     # Recursively resolve all anchors and aliases
658     @staticmethod
659     def resolve_anchors(data):
660         """
661             Recursively resolve all anchors and aliases in a nested data
662             structure.
663
664             This static method traverses through dictionaries and lists,
665             resolving any YAML anchors and aliases.
666
667             Args:
668                 data: The data structure to resolve. Can be a dictionary,
669                 list, or any other type.
670
671             Returns:
672                 The resolved data structure with all anchors and aliases
673                 replaced by their actual values.
674             """
675             if isinstance(data, dict):
676                 return {k: Optimizer.resolve_anchors(v) for k, v in
677                         data.items()}
678             elif isinstance(data, list):
679                 return [Optimizer.resolve_anchors(item) for item in data]
680             else:
681                 return data
682
683     def clean_optimized_config(self) -> dict:
684         """
685             Creates a clean YAML configuration from the optimized operation
686             containers,
687             removing internal fields and organizing operations into proper
688             pipeline steps.
689             """
690             if not self.runner.last_op_container:
691                 return self.config
692
693             # Create a clean copy of the config
694             datasets = {}
695             for dataset_name, dataset_config in self.config.get("datasets",
696             {}).items():
697                 if dataset_config["type"] == "memory":
698                     dataset_config_copy = copy.deepcopy(dataset_config)
699                     dataset_config_copy["path"] = "in-memory data"
700                     datasets[dataset_name] = dataset_config_copy
701                 else:
702                     datasets[dataset_name] = dataset_config
703
704             clean_config = {
705                 "datasets": datasets,
706                 "operations": [],
707                 "pipeline": self.runner.config.get(
708                     "pipeline", {})
709             ).copy(), # Copy entire pipeline config
710         }
711
712         # Reset steps to regenerate
713         clean_config["pipeline"]["steps"] = []
714
```

```

715     # Keep track of operations we've seen to avoid duplicates
716     seen_operations = set()
717
718     def clean_operation(op_container: OpContainer) -> dict:
719         """Remove internal fields from operation config"""
720         op_config = op_container.config
721         clean_op = copy.deepcopy(op_config)
722
723         clean_op.pop("_intermediates", None)
724
725         # If op has already been optimized, remove the
726         # recursively_optimize and optimize fields
727         if op_container.is_optimized:
728             for field in ["recursively_optimize", "optimize"]:
729                 clean_op.pop(field, None)
730
731     return clean_op
732
733     def process_container(container, current_step=None):
734         """Process an operation container and its dependencies"""
735         # Skip step boundaries
736         if isinstance(container, StepBoundary):
737             if container.children:
738                 return process_container(container.children[0],
739                                         current_step)
740
741             return None, None
742
743         # Get step name from container name
744         step_name = container.name.split("/")[0]
745
746         # If this is a new step, create it
747         if not current_step or current_step["name"] != step_name:
748             current_step = {"name": step_name, "operations": []}
749             clean_config["pipeline"]["steps"].insert(0, current_step)
750
751         # Skip scan operations but process their dependencies
752         if container.config["type"] == "scan":
753             if container.children:
754                 return process_container(container.children[0],
755                                         current_step)
756
757             return None, current_step
758
759         # Handle equijoin operations
760         if container.is_equijoin:
761             # Add operation to list if not seen
762             if container.name not in seen_operations:
763                 op_config = clean_operation(container)
764                 clean_config["operations"].append(op_config)
765                 seen_operations.add(container.name)
766
767             # Add to step operations with left and right inputs
768             current_step["operations"].insert(
769                 0,
770                 {
771                     container.config["name"]: {
772                         "left": container.kwargs["left_name"],
773                         "right": container.kwargs["right_name"],
774                     }
775                 },
776             )
777

```

```

        # Process both children
        if container.children:
            process_container(container.children[0],
current_step)
            process_container(container.children[1],
current_step)
        else:
            # Add operation to list if not seen
            if container.name not in seen_operations:
                op_config = clean_operation(container)
                clean_config["operations"].append(op_config)
                seen_operations.add(container.name)

            # Add to step operations
            current_step["operations"].insert(0,
container.config["name"])

        # Process children
        if container.children:
            for child in container.children:
                process_container(child, current_step)

    return container, current_step

# Start processing from the last container
process_container(self.runner.last_op_container)

# Add inputs to steps based on their first operation
for step in clean_config["pipeline"]["steps"]:
    first_op = step["operations"][0]
    if isinstance(first_op, dict): # This is an equijoin
        continue # Equijoin steps don't need an input field
    elif len(step["operations"]) > 0:
        # Find the first non-scan operation's input by looking at
        its dependencies
        op_container = self.runner.op_container_map.get(
            f"{step['name']}/{first_op}"
        )
        if op_container and op_container.children:
            child = op_container.children[0]
            while (
                child
                and child.config["type"] == "step_boundary"
                and child.children
            ):
                child = child.children[0]
            if child and child.config["type"] == "scan":
                step["input"] = child.config["dataset_name"]

    # Preserve all other config key-value pairs from original config
    for key, value in self.config.items():
        if key not in ["datasets", "operations", "pipeline"]:
            clean_config[key] = value

    return clean_config

def save_optimized_config(self, optimized_config_path: str):
    """
    Saves the optimized configuration to a YAML file after resolving
    all references
    and cleaning up internal optimization artifacts.
    """

```

```
"""
resolved_config = self.clean_optimized_config()

with open(optimized_config_path, "w") as f:
    yaml.safe_dump(resolved_config, f, default_flow_style=False,
width=80)
    self.console.log(
        f"[green italic] Optimized config saved to
{optimized_config_path}[/green italic]"
    )

```

```
__init__(runner, rewrite_agent_model='gpt-4o', judge_agent_model='gpt-4o-mini',
litellm_kwargs={}, resume=False, timeout=60)
```

Initialize the optimizer with a runner instance and configuration. Sets up optimization parameters, caching, and cost tracking.

Parameters:

Name	Type	Description	Default
yaml_file	str	Path to the YAML configuration file.	required
model	str	The name of the language model to use. Defaults to "gpt-4o".	required
resume	bool	Whether to resume optimization from a previous run. Defaults to False.	False
timeout	int	Timeout in seconds for operations. Defaults to 60.	60

Attributes:

Name	Type	Description
config	Dict	Stores the loaded configuration from the YAML file.
console	Console	Rich console for formatted output.
max_threads	int	Maximum number of threads for parallel processing.

Name	Type	Description
base_name	str	Base name used for file paths.
yaml_file_suffix	str	Suffix for YAML configuration files.
runner	DSLRunner	The DSL runner instance.
status	DSLRunner	Status tracking for the runner.
optimized_config	Dict	A copy of the original config to be optimized.
llm_client	LLMClient	Client for interacting with the language model.
timeout	int	Timeout for operations in seconds.
resume	bool	Whether to resume from previous optimization.
captured_output	CapturedOutput	Captures output during optimization.
sample_cache	Dict	Maps operation names to tuples of (output_data, sample_size).
optimized_ops_path	str	Path to store optimized operations.
sample_size_map	Dict	Maps operation types to sample sizes.

The method also calls `print_optimizer_config()` to display the initial configuration.

Source code in `docetl/optimizer.py`

```
55     def __init__(  
56         self,  
57         runner: "DSLRunner",  
58         rewrite_agent_model: str = "gpt-4o",  
59         judge_agent_model: str = "gpt-4o-mini",  
60         litellm_kwargs: dict[str, Any] = {},  
61         resume: bool = False,  
62         timeout: int = 60,  
63     ):  
64         """  
65             Initialize the optimizer with a runner instance and configuration.  
66             Sets up optimization parameters, caching, and cost tracking.  
67  
68             Args:  
69                 yaml_file (str): Path to the YAML configuration file.  
70                 model (str): The name of the language model to use. Defaults to  
71 "gpt-4o".  
72                 resume (bool): Whether to resume optimization from a previous  
73 run. Defaults to False.  
74                 timeout (int): Timeout in seconds for operations. Defaults to 60.  
75  
76             Attributes:  
77                 config (Dict): Stores the loaded configuration from the YAML  
78 file.  
79                 console (Console): Rich console for formatted output.  
80                 max_threads (int): Maximum number of threads for parallel  
81 processing.  
82                 base_name (str): Base name used for file paths.  
83                 yaml_file_suffix (str): Suffix for YAML configuration files.  
84                 runner (DSLRunner): The DSL runner instance.  
85                 status: Status tracking for the runner.  
86                 optimized_config (Dict): A copy of the original config to be  
87 optimized.  
88                 llm_client (LLMClient): Client for interacting with the language  
89 model.  
90                 timeout (int): Timeout for operations in seconds.  
91                 resume (bool): Whether to resume from previous optimization.  
92                 captured_output (CapturedOutput): Captures output during  
93 optimization.  
94                 sample_cache (Dict): Maps operation names to tuples of  
95 (output_data, sample_size).  
96                 optimized_ops_path (str): Path to store optimized operations.  
97                 sample_size_map (Dict): Maps operation types to sample sizes.  
98  
99                 The method also calls print_optimizer_config() to display the initial  
100 configuration.  
101             """  
102             self.config = runner.config  
103             self.console = runner.console  
104             self.max_threads = runner.max_threads  
105  
106             self.base_name = runner.base_name  
107             self.yaml_file_suffix = runner.yaml_file_suffix  
108             self.runner = runner  
109             self.status = runner.status  
110  
111             self.optimized_config = copy.deepcopy(self.config)
```

```

112
113     # Get the rate limits from the optimizer config
114     rate_limits = self.config.get("optimizer_config",
115     {}).get("rate_limits", {})
116
117     self.llm_client = LLMClient(
118         runner,
119         rewrite_agent_model,
120         judge_agent_model,
121         rate_limits,
122         **litellm_kwargs,
123     )
124     self.timeout = timeout
125     self.resume = resume
126     self.captured_output = CapturedOutput()
127
128     # Add sample cache for build operations
129     self.sample_cache = {} # Maps operation names to (output_data,
130     sample_size)
131
132     home_dir = os.environ.get("DOCETL_HOME_DIR", os.path.expanduser("~/"))
133     cache_dir = os.path.join(home_dir,
134     f".docetl/cache/{runner.yaml_file_suffix}")
135     os.makedirs(cache_dir, exist_ok=True)

        # Hash the config to create a unique identifier
        config_hash = hashlib.sha256(str(self.config).encode()).hexdigest()
        self.optimized_ops_path = f"{cache_dir}/{config_hash}.yaml"

        # Update sample size map
        self.sample_size_map = SAMPLE_SIZE_MAP
        if self.config.get("optimizer_config", {}).get("sample_sizes", {}):
            self.sample_size_map.update(self.config["optimizer_config"][
                "sample_sizes"])

        if not self.runner._from_df_accessors:
            self.print_optimizer_config()

```

`checkpoint_optimized_ops()`

Generates the clean config and saves it to the `self.optimized_ops_path`. This is used to resume optimization from a previous run

Source code in `docetl/optimizer.py`

```

565     def checkpoint_optimized_ops(self) -> None:
566         """
567             Generates the clean config and saves it to the
568             self.optimized_ops_path
569             This is used to resume optimization from a previous run
570             """
571             clean_config = self.clean_optimized_config()
572             with open(self.optimized_ops_path, "w") as f:
573                 yaml.safe_dump(clean_config, f, default_flow_style=False,
width=80)

```

`clean_optimized_config()`

Creates a clean YAML configuration from the optimized operation containers, removing internal fields and organizing operations into proper pipeline steps.

Source code in `docetl/optimizer.py`

```
595     def clean_optimized_config(self) -> dict:
596         """
597             Creates a clean YAML configuration from the optimized operation
598             containers,
599             removing internal fields and organizing operations into proper
600             pipeline steps.
601             """
602             if not self.runner.last_op_container:
603                 return self.config
604
605             # Create a clean copy of the config
606             datasets = {}
607             for dataset_name, dataset_config in self.config.get("datasets",
608             {}).items():
609                 if dataset_config["type"] == "memory":
610                     dataset_config_copy = copy.deepcopy(dataset_config)
611                     dataset_config_copy["path"] = "in-memory data"
612                     datasets[dataset_name] = dataset_config_copy
613                 else:
614                     datasets[dataset_name] = dataset_config
615
616             clean_config = {
617                 "datasets": datasets,
618                 "operations": [],
619                 "pipeline": self.runner.config.get(
620                     "pipeline", {})
621             ).copy(), # Copy entire pipeline config
622         }
623
624         # Reset steps to regenerate
625         clean_config["pipeline"]["steps"] = []
626
627         # Keep track of operations we've seen to avoid duplicates
628         seen_operations = set()
629
630         def clean_operation(op_container: OpContainer) -> dict:
631             """Remove internal fields from operation config"""
632             op_config = op_container.config
633             clean_op = copy.deepcopy(op_config)
634
635             clean_op.pop("_intermediates", None)
636
637             # If op has already been optimized, remove the
638             # recursively_optimize and optimize fields
639             if op_container.is_optimized:
640                 for field in ["recursively_optimize", "optimize"]:
641                     clean_op.pop(field, None)
642
643             return clean_op
644
645         def process_container(container, current_step=None):
646             """Process an operation container and its dependencies"""
647             # Skip step boundaries
648             if isinstance(container, StepBoundary):
649                 if container.children:
650                     return process_container(container.children[0],
651                     current_step)
```

```

652         return None, None
653
654     # Get step name from container name
655     step_name = container.name.split("/")[-1]
656
657     # If this is a new step, create it
658     if not current_step or current_step["name"] != step_name:
659         current_step = {"name": step_name, "operations": []}
660         clean_config["pipeline"]["steps"].insert(0, current_step)
661
662     # Skip scan operations but process their dependencies
663     if container.config["type"] == "scan":
664         if container.children:
665             return process_container(container.children[0], current_step)
666
667     return None, current_step
668
669     # Handle equijoin operations
670     if container.is_equijoin:
671         # Add operation to list if not seen
672         if container.name not in seen_operations:
673             op_config = clean_operation(container)
674             clean_config["operations"].append(op_config)
675             seen_operations.add(container.name)
676
677         # Add to step operations with left and right inputs
678         current_step["operations"].insert(
679             0,
680             {
681                 container.config["name"]: {
682                     "left": container.kwargs["left_name"],
683                     "right": container.kwargs["right_name"],
684                 }
685             },
686         )
687
688         # Process both children
689         if container.children:
690             process_container(container.children[0], current_step)
691             process_container(container.children[1], current_step)
692         else:
693             # Add operation to list if not seen
694             if container.name not in seen_operations:
695                 op_config = clean_operation(container)
696                 clean_config["operations"].append(op_config)
697                 seen_operations.add(container.name)
698
699             # Add to step operations
700             current_step["operations"].insert(0,
701                 container.config["name"])
702
703             # Process children
704             if container.children:
705                 for child in container.children:
706                     process_container(child, current_step)
707
708     return container, current_step
709
710     # Start processing from the last container
711     process_container(self.runner.last_op_container)
712

```

```

713     # Add inputs to steps based on their first operation
714     for step in clean_config["pipeline"]["steps"]:
715         first_op = step["operations"][0]
716         if isinstance(first_op, dict): # This is an equijoin
717             continue # Equijoin steps don't need an input field
718         elif len(step["operations"]) > 0:
719             # Find the first non-scan operation's input by looking at its
720             dependencies
721             op_container = self.runner.op_container_map.get(
722                 f"{step['name']}/{first_op}"
723             )
724             if op_container and op_container.children:
725                 child = op_container.children[0]
726                 while (
727                     child
728                     and child.config["type"] == "step_boundary"
729                     and child.children
730                 ):
731                     child = child.children[0]
732                 if child and child.config["type"] == "scan":
733                     step["input"] = child.config["dataset_name"]

    # Preserve all other config key-value pairs from original config
    for key, value in self.config.items():
        if key not in ["datasets", "operations", "pipeline"]:
            clean_config[key] = value

    return clean_config

```

`optimize()`

Optimizes the entire pipeline by walking the operation DAG and applying operation-specific optimizers where marked. Returns the total optimization cost.

Source code in `docetl/optimizer.py`

```

418     def optimize(self) -> float:
419         """
420             Optimizes the entire pipeline by walking the operation DAG and
421             applying
422                 operation-specific optimizers where marked. Returns the total
423             optimization cost.
424             """
425             self.console.rule("[bold cyan]Beginning Pipeline Rewrites[/bold
426 cyan]")
427
428             # If self.resume is True and there's a checkpoint, load it
429             if self.resume:
430                 if os.path.exists(self.optimized_ops_path):
431                     # Load the yaml and change the runner with it
432                     with open(self.optimized_ops_path, "r") as f:
433                         partial_optimized_config = yaml.safe_load(f)
434                         self.console.log(
435                             "[yellow]Loading partially optimized pipeline from
436 checkpoint...[/yellow]")
437                     )
438
439             self.runner._build_operation_graph(partial_optimized_config)
440             else:
441                 self.console.log(
442                     "[yellow]No checkpoint found, starting optimization from
443 scratch...[/yellow]")
444             )
445
446             else:
447                 self._insert_empty_resolve_operations()
448
449             # Start with the last operation container and visit each child
450             self.runner.last_op_container.optimize()
451
452             flush_cache(self.console)

                # Print the query plan
                self.console.rule("[bold cyan]Optimized Query Plan[/bold cyan]")
                self.runner.print_query_plan()

            return self.llm_client.total_cost

```

`print_optimizer_config()`

Print the current configuration of the optimizer.

This method uses the Rich console to display a formatted output of the optimizer's configuration. It includes details such as the YAML file path, sample sizes for different operation types, maximum number of threads, the language model being used, and the timeout setting.

The output is color-coded and formatted for easy readability, with a header and separator lines to clearly delineate the configuration information.

Source code in `docetl/optimizer.py`

```

137     def print_optimizer_config(self):
138         """
139             Print the current configuration of the optimizer.
140
141             This method uses the Rich console to display a formatted output of
142             the optimizer's
143                 configuration. It includes details such as the YAML file path, sample
144             sizes for
145                 different operation types, maximum number of threads, the language
146             model being used,
147                 and the timeout setting.
148
149             The output is color-coded and formatted for easy readability, with a
150             header and
151                 separator lines to clearly delineate the configuration information.
152             """
153             self.console.log(
154                 Panel.fit(
155                     "[bold cyan]Optimizer Configuration[/bold cyan]\n"
156                     f"[yellow]Sample Size:{[/yellow]} {self.sample_size_map}\n"
157                     f"[yellow]Max Threads:{[/yellow]} {self.max_threads}\n"
158                     f"[yellow]Rewrite Agent Model:{[/yellow]}"
159                     {self.llm_client.rewrite_agent_model}\n"
160                     f"[yellow]Judge Agent Model:{[/yellow]}"
161                     {self.llm_client.judge_agent_model}\n"
162                     f"[yellow]Rate Limits:{[/yellow]}"
163                     {self.config.get('optimizer_config', {}).get('rate_limits', {})})\n",
164                     title="Optimizer Configuration",
165                 )
166             )

```

`resolve_anchors(data)` staticmethod

Recursively resolve all anchors and aliases in a nested data structure.

This static method traverses through dictionaries and lists, resolving any YAML anchors and aliases.

Parameters:

Name	Type	Description	Default
data		The data structure to resolve. Can be a dictionary, list, or any other type.	<i>required</i>

Returns:

Type	Description
	The resolved data structure with all anchors and aliases replaced by their actual values.

Source code in [docetl/optimizer.py](#)

```
575 @staticmethod
576 def resolve_anchors(data):
577     """
578     Recursively resolve all anchors and aliases in a nested data
579     structure.
580
581     This static method traverses through dictionaries and lists,
582     resolving any YAML anchors and aliases.
583
584     Args:
585         data: The data structure to resolve. Can be a dictionary, list,
586     or any other type.
587
588     Returns:
589         The resolved data structure with all anchors and aliases replaced
590     by their actual values.
591     """
592     if isinstance(data, dict):
593         return {k: Optimizer.resolve_anchors(v) for k, v in data.items()}
594     elif isinstance(data, list):
595         return [Optimizer.resolve_anchors(item) for item in data]
596     else:
597         return data
```

`save_optimized_config(optimized_config_path)`

Saves the optimized configuration to a YAML file after resolving all references and cleaning up internal optimization artifacts.

Source code in `docetl/optimizer.py`

```
734     def save_optimized_config(self, optimized_config_path: str):
735         """
736             Saves the optimized configuration to a YAML file after resolving all
737             references
738             and cleaning up internal optimization artifacts.
739             """
740             resolved_config = self.clean_optimized_config()
741
742             with open(optimized_config_path, "w") as f:
743                 yaml.safe_dump(resolved_config, f, default_flow_style=False,
744 width=80)
745                 self.console.log(
746                     f"[green italic][H] Optimized config saved to
{optimized_config_path}[/green italic]"
747                 )
```

`should_optimize(step_name, op_name)`

Analyzes whether an operation should be optimized by running it on a sample of input data and evaluating potential optimizations. Returns the optimization suggestion and relevant data.

Source code in `docetl/optimizer.py`

```
346     def should_optimize(
347         self, step_name: str, op_name: str
348     ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]], float]:
349         """
350             Analyzes whether an operation should be optimized by running it on a
351             sample of input data
352             and evaluating potential optimizations. Returns the optimization
353             suggestion and relevant data.
354         """
355         self.console.rule("[bold cyan]Beginning Pipeline Assessment[/bold"
356         cyan]")
357
358         self._insert_empty_resolve_operations()
359
360         node_of_interest = self.runner.op_container_map[f"
361         {step_name}/{op_name}"]
362
363         # Run the node_of_interest's children
364         input_data = []
365         for child in node_of_interest.children:
366             input_data.append(
367                 child.next(
368                     is_build=True,
369
370                     sample_size_needed=SAMPLE_SIZE_MAP.get(child.config["type"]),
371                     )[0]
372             )
373
374         # Set the step
375         self.captured_output.set_step(step_name)
376
377         # Determine whether we should optimize the node_of_interest
378         if (
379             node_of_interest.config.get("type") == "map"
380             or node_of_interest.config.get("type") == "filter"
381         ):
382             # Create instance of map optimizer
383             map_optimizer = MapOptimizer(
384                 self.runner,
385                 self.runner._run_operation,
386                 is_filter=node_of_interest.config.get("type") == "filter",
387             )
388             should_optimize_output, input_data, output_data = (
389                 map_optimizer.should_optimize(node_of_interest.config,
390                 input_data[0])
391             )
392             elif node_of_interest.config.get("type") == "reduce":
393                 reduce_optimizer = ReduceOptimizer(
394                     self.runner,
395                     self.runner._run_operation,
396                 )
397                 should_optimize_output, input_data, output_data = (
398                     reduce_optimizer.should_optimize(node_of_interest.config,
399                     input_data[0])
400                 )
401             elif node_of_interest.config.get("type") == "resolve":
402                 resolve_optimizer = JoinOptimizer(
```

```
403         self.runner,
404         node_of_interest.config,
405         target_recall=self.config.get("optimizer_config", {})
406         .get("resolve", {})
407         .get("target_recall", 0.95),
408     )
409     _, should_optimize_output =
410 resolve_optimizer.should_optimize(input_data[0])
411
412     # if should_optimize_output is empty, then we should move to the
413     # reduce operation
414     if should_optimize_output == "":
415         return "", [], [], 0.0
416     else:
417         return "", [], [], 0.0
418
419     # Return the string and operation cost
420     return (
421         should_optimize_output,
422         input_data,
423         output_data,
424         self.runner.total_cost + self.llm_client.total_cost,
425     )
```

CLI Interface

```
docetl.cli.run(yaml_file=typer.Argument(..., help='Path to the  
YAML file containing the pipeline configuration'),  
max_threads=typer.Option(None, help='Maximum number of threads to  
use for running operations'))
```

Run the configuration specified in the YAML file.

Parameters:

Name	Type	Description	Default
yaml_file	Path	Path to the YAML file containing the pipeline configuration.	Argument(..., help='Path to the YAML file containing the pipeline configuration')
max_threads	int None	Maximum number of threads to use for running operations.	Option(None, help='Maximum number of threads to use for running operations')

Source code in `docetl/cli.py`

```

56     @app.command()
57     def run(
58         yaml_file: Path = typer.Argument(
59             ...,
60             help="Path to the YAML file containing the pipeline
61             configuration"
62         ),
63         max_threads: int | None = typer.Option(
64             None,
65             help="Maximum number of threads to use for running
66             operations"
67         ),
68     ):
69         """
70         Run the configuration specified in the YAML file.
71
72         Args:
73             yaml_file (Path): Path to the YAML file containing the pipeline
74             configuration.
75             max_threads (int | None): Maximum number of threads to use for
76             running operations.
77         """
78         # Get the current working directory (where the user called the
79         # command)
80         cwd = os.getcwd()
81
82         # Load .env file from the current working directory
83         env_file = os.path.join(cwd, ".env")
84         if os.path.exists(env_file):
85             load_dotenv(env_file)
86
87         runner = DSLRunner.from_yaml(str(yaml_file), max_threads=max_threads)
88         runner.load_run_save()

```

```

docetl.cli.build(yaml_file=typer.Argument(..., help='Path to the
YAML file containing the pipeline configuration'),
max_threads=typer.Option(None, help='Maximum number of threads to
use for running operations'), resume=typer.Option(False,
help='Resume optimization from a previous build that may have
failed'), save_path=typer.Option(None, help='Path to save the
optimized pipeline configuration'))

```

Build and optimize the configuration specified in the YAML file. Any arguments passed here will override the values in the YAML file.

Parameters:

Name	Type	Description	Default
yaml_file	Path	Path to the YAML file containing the pipeline configuration.	Argument(..., help='Path to the YAML file containing the pipeline configuration')
max_threads	int None	Maximum number of threads to use for running operations.	Option(None, help='Maximum number of threads to use for running operations')
model	str	Model to use for optimization. Defaults to "gpt-4o".	<i>required</i>
resume	bool	Whether to resume optimization from a previous run. Defaults to False.	Option(False, help='Resume optimization from a previous build that may have failed')
save_path	Path	Path to save the optimized pipeline configuration.	Option(None, help='Path to save the optimized pipeline configuration')

Source code in `docetl/cli.py`

```

13     @app.command()
14     def build(
15         yaml_file: Path = typer.Argument(
16             ...,
17             help="Path to the YAML file containing the pipeline
18             configuration"
19         ),
20         max_threads: int | None = typer.Option(
21             None,
22             help="Maximum number of threads to use for running
23             operations"
24         ),
25         resume: bool = typer.Option(
26             False,
27             help="Resume optimization from a previous build that may
28             have failed"
29         ),
30     ):
31         """
32             Build and optimize the configuration specified in the YAML file.
33             Any arguments passed here will override the values in the YAML file.
34
35             Args:
36                 yaml_file (Path): Path to the YAML file containing the pipeline
37                 configuration.
38                 max_threads (int | None): Maximum number of threads to use for
39                 running operations.
40                 model (str): Model to use for optimization. Defaults to "gpt-4o".
41                 resume (bool): Whether to resume optimization from a previous run.
42             Defaults to False.
43                 save_path (Path): Path to save the optimized pipeline
44                 configuration.
45             """
46             # Get the current working directory (where the user called the
47             command)
48             cwd = os.getcwd()
49
50             # Load .env file from the current working directory
51             env_file = os.path.join(cwd, ".env")
52             if os.path.exists(env_file):
53                 load_dotenv(env_file)
54
55             runner = DSLRunner.from_yaml(str(yaml_file), max_threads=max_threads)
56             runner.optimize(
57                 save=True,
58                 return_pipeline=False,
59                 resume=resume,
60                 save_path=save_path,
61             )

```

`docetl.cli.clear_cache()`

Clear the LLM cache stored on disk.

Source code in `docetl/cli.py`

```
84 |     @app.command()
85 |     def clear_cache():
86 |         """
87 |             Clear the LLM cache stored on disk.
88 |         """
89 |         cc()
```

LLM-Powered Operators

`docetl.operations.map.MapOperation`

Bases: `BaseOperation`

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

```
22     class MapOperation(BaseOperation):
23         class schema(BaseOperation.schema):
24             type: str = "map"
25             output: dict[str, Any] | None = None
26             prompt: str | None = None
27             model: str | None = None
28             optimize: bool | None = None
29             recursively_optimize: bool | None = None
30             sample_size: int | None = None
31             tools: list[dict[str, Any]] | None = (
32                 None # FIXME: Why isn't this using the Tool data class so
33                 validation works automatically?
34             )
35             validation_rules: list[str] | None = Field(None,
36             alias="validate")
37             num_retries_on_validate_failure: int | None = None
38             drop_keys: list[str] | None = None
39             timeout: int | None = None
40             enable_observability: bool = False
41             batch_size: int | None = None
42             clustering_method: str | None = None
43             batch_prompt: str | None = None
44             litellm_completion_kwargs: dict[str, Any] = {}
45             pdf_url_key: str | None = None
46             flush_partial_result: bool = False
47             # Calibration parameters
48             calibrate: bool = False
49             num_calibration_docs: int = Field(10, gt=0)
50
51             @field_validator("batch_prompt")
52             def validate_batch_prompt(cls, v):
53                 if v is not None:
54                     try:
55                         template = Template(v)
56                         # Test render with a minimal inputs list to validate
57                         template
58                         template.render(inputs=[{}])
59                     except Exception as e:
60                         raise ValueError(
61                             f"Invalid Jinja2 template in 'batch_prompt' or
62                             missing required 'inputs' variable: {str(e)}")
63                     ) from e
64                 return v
65
66             @field_validator("prompt")
67             def validate_prompt(cls, v):
68                 if v is not None:
69                     try:
70                         Template(v)
71                     except Exception as e:
72                         raise ValueError(
73                             f"Invalid Jinja2 template in 'prompt': {str(e)}")
74                     ) from e
75                 return v
76
77             @field_validator("tools")
78             def validate_tools(cls, v):
```

```

79         if v is not None:
80             for tool in v:
81                 try:
82                     tool_obj = Tool(**tool)
83                 except Exception:
84                     raise TypeError("Tool must be a dictionary")
85
86             if not (tool_obj.code and tool_obj.function):
87                 raise ValueError(
88                     "Tool is missing required 'code' or "
89                     "'function' key"
90                 )
91
92             if not isinstance(tool_obj.function, ToolFunction):
93                 raise TypeError("'function' in tool must be a "
94                     "dictionary")
95
96             for key in ["name", "description", "parameters"]:
97                 if not hasattr(tool_obj.function, key):
98                     raise ValueError(
99                         f"Tool is missing required '{key}' in "
100                        "'function'"
101                    )
102
103
104     @model_validator(mode="after")
105     def validate_prompt_and_output_requirements(self):
106         # If drop_keys is not specified, both prompt and output must
107         be present
108         if not self.drop_keys:
109             if not self.prompt or not self.output:
110                 raise ValueError(
111                     "If 'drop_keys' is not specified, both 'prompt' "
112                     "and 'output' must be present in the configuration"
113                 )
114
115         if self.output and not self.output.get("schema"):
116             raise ValueError("Missing 'schema' in 'output' "
117                     "configuration")
118
119         return self
120
121     def __init__(
122         self,
123         *args,
124         **kwargs,
125     ):
126         super().__init__(*args, **kwargs)
127         self.max_batch_size: int = self.config.get(
128             "max_batch_size", kwargs.get("max_batch_size", None)
129         )
130         self.clustering_method = "random"
131
132     def _generate_calibration_context(self, input_data: list[dict]) ->
133     str:
134         """
135             Generate calibration context by running the operation on a sample
136             of documents
137             and using an LLM to suggest prompt improvements for consistency.
138
139             Returns:

```

```

140         str: Additional context to add to the original prompt
141         """
142         import random
143
144         # Set seed for reproducibility
145         random.seed(42)
146
147         # Sample documents for calibration
148         num_calibration_docs = min(
149             self.config.get("num_calibration_docs", 10), len(input_data)
150         )
151         if num_calibration_docs == len(input_data):
152             calibration_sample = input_data
153         else:
154             calibration_sample = random.sample(input_data,
155             num_calibration_docs)
156
157         self.console.log(
158             f"[bold blue]Running calibration on {num_calibration_docs} "
159             "documents...[/bold blue]"
160         )
161
162         # Temporarily disable calibration to avoid infinite recursion
163         original_calibrate = self.config.get("calibrate", False)
164         self.config["calibrate"] = False
165
166         try:
167             # Run the map operation on the calibration sample
168             calibration_results, _ = self.execute(calibration_sample)
169
170             # Prepare the calibration analysis prompt
171             calibration_prompt = """
172             The following prompt was applied to sample documents to generate these
173             input-output pairs:
174
175             "{self.config["prompt"]}"
176
177             Sample inputs and their outputs:
178             """
179
180             for i, (input_doc, output_doc) in enumerate(
181                 zip(calibration_sample, calibration_results)
182             ):
183                 calibration_prompt += f"\n--- Example {i+1} ---\n"
184                 calibration_prompt += f"Input: {input_doc}\n"
185                 calibration_prompt += f"Output: {output_doc}\n"
186
187             calibration_prompt += """
188             Based on these examples, provide reference anchors that will be appended
189             to the prompt to help maintain consistency when processing all documents.
190
191             DO NOT provide generic advice. Instead, use specific examples from above
192             as calibration points.
193             Note that the outputs might be incorrect, because the user's prompt was
194             not calibrated or rich in the first place.
195             You can ignore the outputs if they are incorrect, and focus on the
196             diversity of the inputs.
197
198             Format as concrete reference points:
199             - "For reference, consider '[specific input text]' → [output] as a
200               baseline for [category/level]"
```

```

201     - "Documents similar to '[specific input text]' should be classified as
202     [output]"'
203
204     Reference anchors:''''"
205
206         # Call LLM to get calibration suggestions
207         messages = [{"role": "user", "content": calibration_prompt}]
208             # Use a copy of the user-provided completion kwargs so we
209             don't mutate the original
210                 # and avoid hard-coding temperature to a value that may not
211                 be supported by certain models.
212                     completion_kwargs =
213                         dict(self.config.get("litellm_completion_kwargs", {}))
214                             # If the user did not explicitly specify a temperature, let
215                             the model default handle it
216                                 # to prevent incompatibility errors with providers that don't
217                                 support 0.0.
218                                     # If a temperature is already provided, respect the user's
219                                     choice.
220
221             llm_result = self.runner.api.call_llm(
222                 self.config.get("model", self.default_model),
223                 "calibration",
224                 messages,
225                 {"calibration_context": "string"},,
226                 timeout_seconds=self.config.get("timeout", 120),
227
228             max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
229                 bypass_cache=self.config.get("bypass_cache",
230 self.bypass_cache),
231                     litellm_completion_kwargs=completion_kwargs,
232                     op_config=self.config,
233             )
234
235             # Parse the response
236             if hasattr(llm_result, "response"):
237                 calibration_context = self.runner.api.parse_llm_response(
238                     llm_result.response,
239                     schema={"calibration_context": "string"},,
240                     manually_fix_errors=self.manually_fix_errors,
241                     )[0].get("calibration_context", "")
242             else:
243                 calibration_context = ""
244
245             return calibration_context
246
247         finally:
248             # Restore original calibration setting
249             self.config["calibrate"] = original_calibrate
250
251     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
252         """
253             Executes the map operation on the provided input data.
254
255             Args:
256                 input_data (list[dict]): The input data to process.
257
258             Returns:
259                 tuple[list[dict], float]: A tuple containing the processed
260                 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
281     dropped_results = []
282     for item in input_data:
283         new_item = {
284             k: v for k, v in item.items() if k not in
285             self.config["drop_keys"]
286         }
287         dropped_results.append(new_item)
288     return dropped_results, 0.0 # Return the modified data with
289     no cost
290
291     # Generate calibration context if enabled
292     calibration_context = ""
293     if self.config.get("calibrate", False) and "prompt" in
294     self.config:
295         calibration_context =
296         self._generate_calibration_context(input_data)
297         if calibration_context:
298             # Store original prompt for potential restoration
299             self._original_prompt = self.config["prompt"]
300             # Augment the prompt with calibration context
301             self.config["prompt"] =
302                 f"{self.config['prompt']}\n\n{calibration_context}"
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":
```

```

323     item})
324         messages = [{"role": "user", "content": prompt}]
325     if self.config.get("pdf_url_key", None):
326         # Append the pdf to the prompt
327         try:
328             pdf_url = item[self.config["pdf_url_key"]]
329         except KeyError:
330             raise ValueError(
331                 f"PDF URL key '{self.config['pdf_url_key']}' not
332 found in input data"
333             )
334
335         # Download content
336         if pdf_url.startswith("http"):
337             file_data = requests.get(pdf_url).content
338         else:
339             with open(pdf_url, "rb") as f:
340                 file_data = f.read()
341             encoded_file = base64.b64encode(file_data).decode("utf-
342 8")
343             base64_url = f"data:application/pdf;base64,
344 {encoded_file}"
345
346             messages[0]["content"] = [
347                 {"type": "image_url", "image_url": {"url"::
348 base64_url}},
349                 {"type": "text", "text": prompt},
350             ]
351
352     def validation_fn(response: dict[str, Any] | ModelResponse):
353         structured_mode = (
354             self.config.get("output", {}).get("mode")
355             == OutputMode.STRUCTURED_OUTPUT.value
356         )
357         output = (
358             self.runner.api.parse_llm_response(
359                 response,
360                 schema=self.config["output"]["schema"],
361                 tools=self.config.get("tools", None),
362                 manually_fix_errors=self.manually_fix_errors,
363                 use_structured_output=structured_mode,
364             )[0]
365             if isinstance(response, ModelResponse)
366             else response
367         )
368         # Check that the output has all the keys in the schema
369         for key in self.config["output"]["schema"]:
370             if key not in output:
371                 return output, False
372
373             for key, value in item.items():
374                 if key not in self.config["output"]["schema"]:
375                     output[key] = value
376                 if self.runner.api.validate_output(self.config, output,
377 self.console):
378                     return output, True
379             return output, False
380
381             if self.runner.is_cancelled:
382                 raise asyncio.CancelledError("Operation was cancelled")
383             llm_result = self.runner.api.call_llm(

```

```

384         self.config.get("model", self.default_model),
385         "map",
386         messages,
387         self.config["output"]["schema"],
388         tools=self.config.get("tools", None),
389         scratchpad=None,
390         timeout_seconds=self.config.get("timeout", 120),
391
392     max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
393     validation_config=(
394         {
395             "num_retries":
396             self.num_retries_on_validate_failure,
397             "val_rule": self.config.get("validate", []),
398             "validation_fn": validation_fn,
399         }
400         if self.config.get("validate", None)
401         else None
402     ),
403     gleaning_config=self.config.get("gleaning", None),
404     verbose=self.config.get("verbose", False),
405     bypass_cache=self.config.get("bypass_cache",
406     self.bypass_cache),
407     initial_result=initial_result,
408     litellm_completion_kwargs=self.config.get(
409         "litellm_completion_kwargs", {}
410     ),
411     op_config=self.config,
412 )
413
414 if llm_result.validated:
415     # Parse the response
416     if isinstance(llm_result.response, ModelResponse):
417         structured_mode = (
418             self.config.get("output", {}).get("mode")
419             == OutputMode.STRUCTURED_OUTPUT.value
420         )
421         outputs = self.runner.api.parse_llm_response(
422             llm_result.response,
423             schema=self.config["output"]["schema"],
424             tools=self.config.get("tools", None),
425             manually_fix_errors=self.manually_fix_errors,
426             use_structured_output=structured_mode,
427         )
428     else:
429         outputs = [llm_result.response]
430
431     # Augment the output with the original item
432     outputs = [{**item, **output} for output in outputs]
433     if self.config.get("enable_observability", False):
434         for output in outputs:
435             output[f"_observability_{self.config['name']}"] =
436             {
437                 "prompt": prompt
438             }
439     return outputs, llm_result.total_cost
440
441     return None, llm_result.total_cost
442
443     # If there's a batch prompt, let's use that
444     def _process_map_batch(items: list[dict]) -> tuple[list[dict],

```

```
445     float]:
446         total_cost = 0
447         if len(items) > 1 and self.config.get("batch_prompt", None):
448             # Raise error if pdf_url_key is set
449             if self.config.get("pdf_url_key", None):
450                 raise ValueError("Batch prompts do not support PDF
451 URLs")
452
453         batch_prompt = strict_render(
454             self.config["batch_prompt"], {"inputs": items}
455         )
456
457         # Issue the batch call
458         llm_result = self.runner.api.call_llm_batch(
459             self.config.get("model", self.default_model),
460             "batch_map",
461             [{"role": "user", "content": batch_prompt}],
462             self.config["output"]["schema"],
463             verbose=self.config.get("verbose", False),
464             timeout_seconds=self.config.get("timeout", 120),
465             max_retries_per_timeout=self.config.get(
466                 "max_retries_per_timeout", 2
467             ),
468             bypass_cache=self.config.get("bypass_cache",
469             self.bypass_cache),
470             litellm_completion_kwargs=self.config.get(
471                 "litellm_completion_kwargs", {}
472             ),
473         )
474         total_cost += llm_result.total_cost
475
476         # Parse the LLM response
477         structured_mode = (
478             self.config.get("output", {}).get("mode")
479             == OutputMode.STRUCTURED_OUTPUT.value
480         )
481         parsed_output = self.runner.api.parse_llm_response(
482             llm_result.response,
483             self.config["output"]["schema"],
484             use_structured_output=structured_mode,
485         )[0].get("results", [])
486         items_and_outputs = [
487             (item, parsed_output[idx] if idx < len(parsed_output)
488             else None)
489             for idx, item in enumerate(items)
490         ]
491     else:
492         items_and_outputs = [(item, None) for item in items]
493
494         # Run _process_map_item for each item
495         all_results = []
496         if len(items_and_outputs) > 1:
497             with ThreadPoolExecutor(max_workers=self.max_batch_size)
498             as executor:
499                 futures = [
500                     executor.submit(
501                         _process_map_item,
502                         items_and_outputs[i][0],
503                         items_and_outputs[i][1],
504                     )
505                     for i in range(len(items_and_outputs))]
```

```

506             ]
507         for i in range(len(futures)):
508             try:
509                 results, item_cost = futures[i].result()
510                 if results is not None:
511                     all_results.extend(results)
512                     total_cost += item_cost
513             except Exception as e:
514                 if self.config.get("skip_on_error", False):
515                     self.console.log(
516                         f"[bold red]Error in map operation
517 {self.config['name']}], skipping item:{[/bold red] {e}""
518                         )
519                     continue
520             else:
521                 raise e
522
523     else:
524         try:
525             results, item_cost = _process_map_item(
526                 items_and_outputs[0][0], items_and_outputs[0][1]
527             )
528             if results is not None:
529                 all_results.extend(results)
530                 total_cost += item_cost
531         except Exception as e:
532             if self.config.get("skip_on_error", False):
533                 self.console.log(
534                     f"[bold red]Error in map operation
535 {self.config['name']}], skipping item:{[/bold red] {e}""
536                     )
537             else:
538                 raise e
539
540
541     return all_results, total_cost
542
543
544     with ThreadPoolExecutor(max_workers=self.max_batch_size) as
545 executor:
546     batch_size = self.max_batch_size if self.max_batch_size is
547 not None else 1
548     futures = []
549     for i in range(0, len(input_data), batch_size):
550         batch = input_data[i : i + batch_size]
551         futures.append(executor.submit(_process_map_batch,
batch))
552
553     results = []
554     total_cost = 0
555     pbar = RichLoopBar(
556         range(len(futures)),
557         desc=f"Processing {self.config['name']} (map) on all
558 documents",
559         console=self.console,
560     )
561     for batch_index in pbar:
562         result_list, item_cost = futures[batch_index].result()
563         if result_list:
564             if "drop_keys" in self.config:
565                 result_list = [
566                     {
567                         k: v
568                         for k, v in result.items()
569                         if k not in self.config["drop_keys"]
570                     }
571                 ]
572
573     return all_results, total_cost

```

```

        }
        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.	<i>required</i>

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`

```
222     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:  
223         """  
224             Executes the map operation on the provided input data.  
225  
226             Args:  
227                 input_data (list[dict]): The input data to process.  
228  
229             Returns:  
230                 tuple[list[dict], float]: A tuple containing the processed  
231             results and the total cost of the operation.  
232  
233             This method performs the following steps:  
234                 1. If calibration is enabled, runs calibration to improve prompt  
consistency  
236                 2. If a prompt is specified, it processes each input item using the  
237             specified prompt and LLM model  
238                     3. Applies gleaning if configured  
239                     4. Validates the output  
240                     5. If drop_keys is specified, it drops the specified keys from each  
241             document  
242                     6. Aggregates results and calculates total cost  
243  
244             The method uses parallel processing to improve performance.  
245         """  
246  
247         # Check if there's no prompt and only drop_keys  
248         if "prompt" not in self.config and "drop_keys" in self.config:  
249             # If only drop_keys is specified, simply drop the keys and return  
250             dropped_results = []  
251             for item in input_data:  
252                 new_item = {  
253                     k: v for k, v in item.items() if k not in  
self.config["drop_keys"]  
254                 }  
255                 dropped_results.append(new_item)  
256             return dropped_results, 0.0 # Return the modified data with no  
cost  
258  
259             # Generate calibration context if enabled  
260             calibration_context = ""  
261             if self.config.get("calibrate", False) and "prompt" in self.config:  
262                 calibration_context =  
263                 self._generate_calibration_context(input_data)  
264                 if calibration_context:  
265                     # Store original prompt for potential restoration  
266                     self._original_prompt = self.config["prompt"]  
267                     # Augment the prompt with calibration context  
268                     self.config["prompt"] = (  
269                         f"{self.config['prompt']}\\n\\n{calibration_context}"  
270                     )  
271                     self.console.log(  
272                         f"[bold green]New map ({self.config['name']}) prompt  
273                         augmented with context on how to improve consistency:[/bold green]  
274                         {self.config['prompt']}"  
275                     )  
276                 else:  
277                     self.console.log(  
278                         f"[bold yellow]Extra context on how to improve
```

```

279     consistency failed to generate for map ({self.config['name']});  

280     continuing with prompt as is.[/bold yellow]"  

281         )  

282  

283     if self.status:  

284         self.status.stop()  

285  

286     def _process_map_item(  

287         item: dict, initial_result: dict | None = None  

288     ) -> tuple[dict | None, float]:  

289  

290         prompt = strict_render(self.config["prompt"], {"input": item})  

291         messages = [{"role": "user", "content": prompt}]  

292         if self.config.get("pdf_url_key", None):  

293             # Append the pdf to the prompt  

294             try:  

295                 pdf_url = item[self.config["pdf_url_key"]]  

296             except KeyError:  

297                 raise ValueError(  

298                     f"PDF URL key '{self.config['pdf_url_key']}' not  

299 found in input data"  

300             )  

301  

302             # Download content  

303             if pdf_url.startswith("http"):  

304                 file_data = requests.get(pdf_url).content  

305             else:  

306                 with open(pdf_url, "rb") as f:  

307                     file_data = f.read()  

308             encoded_file = base64.b64encode(file_data).decode("utf-8")  

309             base64_url = f"data:application/pdf;base64,{encoded_file}"  

310  

311             messages[0]["content"] = [  

312                 {"type": "image_url", "image_url": {"url": base64_url}},  

313                 {"type": "text", "text": prompt},  

314             ]  

315  

316     def validation_fn(response: dict[str, Any] | ModelResponse):  

317         structured_mode = (  

318             self.config.get("output", {}).get("mode")  

319             == OutputMode.STRUCTURED_OUTPUT.value  

320         )  

321         output = (  

322             self.runner.api.parse_llm_response(  

323                 response,  

324                 schema=self.config["output"]["schema"],  

325                 tools=self.config.get("tools", None),  

326                 manually_fix_errors=self.manually_fix_errors,  

327                 use_structured_output=structured_mode,  

328             )[0]  

329             if isinstance(response, ModelResponse)  

330             else response  

331         )  

332         # Check that the output has all the keys in the schema  

333         for key in self.config["output"]["schema"]:  

334             if key not in output:  

335                 return output, False  

336  

337             for key, value in item.items():  

338                 if key not in self.config["output"]["schema"]:  

339                     output[key] = value

```

```

340         if self.runner.api.validate_output(self.config, output,
341             self.console):
342             return output, True
343             return output, False
344
345     if self.runner.is_cancelled:
346         raise asyncio.CancelledError("Operation was cancelled")
347     llm_result = self.runner.api.call_llm(
348         self.config.get("model", self.default_model),
349         "map",
350         messages,
351         self.config["output"]["schema"],
352         tools=self.config.get("tools", None),
353         scratchpad=None,
354         timeout_seconds=self.config.get("timeout", 120),
355
356     max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
357     validation_config=(
358         {
359             "num_retries": self.num_retries_on_validate_failure,
360             "val_rule": self.config.get("validate", []),
361             "validation_fn": validation_fn,
362         }
363         if self.config.get("validate", None)
364         else None
365     ),
366     gleaning_config=self.config.get("gleaning", None),
367     verbose=self.config.get("verbose", False),
368     bypass_cache=self.config.get("bypass_cache",
369     self.bypass_cache),
370     initial_result=initial_result,
371     litellm_completion_kwargs=self.config.get(
372         "litellm_completion_kwargs", {}
373     ),
374     op_config=self.config,
375 )
376
377     if llm_result.validated:
378         # Parse the response
379         if isinstance(llm_result.response, ModelResponse):
380             structured_mode = (
381                 self.config.get("output", {}).get("mode")
382                 == OutputMode.STRUCTURED_OUTPUT.value
383             )
384             outputs = self.runner.api.parse_llm_response(
385                 llm_result.response,
386                 schema=self.config["output"]["schema"],
387                 tools=self.config.get("tools", None),
388                 manually_fix_errors=self.manually_fix_errors,
389                 use_structured_output=structured_mode,
390             )
391         else:
392             outputs = [llm_result.response]
393
394         # Augment the output with the original item
395         outputs = [{**item, **output} for output in outputs]
396         if self.config.get("enable_observability", False):
397             for output in outputs:
398                 output[f"_observability_{self.config['name']}"] = {
399                     "prompt": prompt
400                 }

```

```

401         return outputs, llm_result.total_cost
402
403     return None, llm_result.total_cost
404
405     # If there's a batch prompt, let's use that
406     def _process_map_batch(items: list[dict]) -> tuple[list[dict],
407     float]:
407
408         total_cost = 0
409         if len(items) > 1 and self.config.get("batch_prompt", None):
410             # Raise error if pdf_url_key is set
411             if self.config.get("pdf_url_key", None):
412                 raise ValueError("Batch prompts do not support PDF URLs")
413
414             batch_prompt = strict_render(
415                 self.config["batch_prompt"], {"inputs": items}
416             )
417
418             # Issue the batch call
419             llm_result = self.runner.api.call_llm_batch(
420                 self.config.get("model", self.default_model),
421                 "batch_map",
422                 [{"role": "user", "content": batch_prompt}],
423                 self.config["output"]["schema"],
424                 verbose=self.config.get("verbose", False),
425                 timeout_seconds=self.config.get("timeout", 120),
426                 max_retries_per_timeout=self.config.get(
427                     "max_retries_per_timeout", 2
428                 ),
429                 bypass_cache=self.config.get("bypass_cache",
430                 self.bypass_cache),
431                 litellm_completion_kwargs=self.config.get(
432                     "litellm_completion_kwargs", {}
433                 ),
434             )
435             total_cost += llm_result.total_cost
436
437             # Parse the LLM response
438             structured_mode = (
439                 self.config.get("output", {}).get("mode")
440                 == OutputMode.STRUCTURED_OUTPUT.value
441             )
442             parsed_output = self.runner.api.parse_llm_response(
443                 llm_result.response,
444                 self.config["output"]["schema"],
445                 use_structured_output=structured_mode,
446             )[0].get("results", [])
447             items_and_outputs = [
448                 (item, parsed_output[idx] if idx < len(parsed_output)
449             else None)
450                 for idx, item in enumerate(items)
451             ]
452         else:
453             items_and_outputs = [(item, None) for item in items]
454
455             # Run _process_map_item for each item
456             all_results = []
457             if len(items_and_outputs) > 1:
458                 with ThreadPoolExecutor(max_workers=self.max_batch_size) as
459                 executor:
460                     futures = [
461                         executor.submit(

```

```

462             _process_map_item,
463             items_and_outputs[i][0],
464             items_and_outputs[i][1],
465         )
466     for i in range(len(items_and_outputs))
467 ]
468 for i in range(len(futures)):
469     try:
470         results, item_cost = futures[i].result()
471         if results is not None:
472             all_results.extend(results)
473             total_cost += item_cost
474     except Exception as e:
475         if self.config.get("skip_on_error", False):
476             self.console.log(
477                 f"[bold red]Error in map operation
478 {self.config['name']}, skipping item:{[bold red] {e}}"
479             )
480             continue
481         else:
482             raise e
483     else:
484         try:
485             results, item_cost = _process_map_item(
486                 items_and_outputs[0][0], items_and_outputs[0][1]
487             )
488             if results is not None:
489                 all_results.extend(results)
490                 total_cost += item_cost
491         except Exception as e:
492             if self.config.get("skip_on_error", False):
493                 self.console.log(
494                     f"[bold red]Error in map operation
495 {self.config['name']}, skipping item:{[bold red] {e}}"
496                 )
497             else:
498                 raise e
499
500     return all_results, total_cost
501
502     with ThreadPoolExecutor(max_workers=self.max_batch_size) as executor:
503         batch_size = self.max_batch_size if self.max_batch_size is not
504         None else 1
505         futures = []
506         for i in range(0, len(input_data), batch_size):
507             batch = input_data[i : i + batch_size]
508             futures.append(executor.submit(_process_map_batch, batch))
509         results = []
510         total_cost = 0
511         pbar = RichLoopBar(
512             range(len(futures)),
513             desc=f"Processing {self.config['name']} (map) on all
514             documents",
515             console=self.console,
516         )
517         for batch_index in pbar:
518             result_list, item_cost = futures[batch_index].result()
519             if result_list:
520                 if "drop_keys" in self.config:
521                     result_list = [
522

```

```
        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

```
27  class ResolveOperation(BaseOperation):
28      class schema(BaseOperation.schema):
29          type: str = "resolve"
30          comparison_prompt: str
31          resolution_prompt: str | None = None
32          output: dict[str, Any] | None = None
33          embedding_model: str | None = None
34          resolution_model: str | None = None
35          comparison_model: str | None = None
36          blocking_keys: list[str] | None = None
37          blocking_threshold: float | None = Field(None, ge=0, le=1)
38          blocking_conditions: list[str] | None = None
39          input: dict[str, Any] | None = None
40          embedding_batch_size: int | None = Field(None, gt=0)
41          compare_batch_size: int | None = Field(None, gt=0)
42          limit_comparisons: int | None = Field(None, gt=0)
43          optimize: bool | None = None
44          timeout: int | None = Field(None, gt=0)
45          litellm_completion_kwargs: dict[str, Any] =
46          Field(default_factory=dict)
47          enable_observability: bool = False
48
49          @field_validator("comparison_prompt")
50          def validate_comparison_prompt(cls, v):
51              if v is not None:
52                  try:
53                      comparison_template = Template(v)
54                      comparison_vars =
55                      comparison_template.environment.parse(v).find_all(
56                          jinja2.nodes.Name
57                      )
58                      comparison_var_names = {var.name for var in
59                      comparison_vars}
60                      if (
61                          "input1" not in comparison_var_names
62                          or "input2" not in comparison_var_names
63                      ):
64                          raise ValueError(
65                              f"'comparison_prompt' must contain both
66 'input1' and 'input2' variables. {v}"
67                          )
68                  except Exception as e:
69                      raise ValueError(
70                          f"Invalid Jinja2 template in 'comparison_prompt':
71 {str(e)}"
72                      )
73              return v
74
75          @field_validator("resolution_prompt")
76          def validate_resolution_prompt(cls, v):
77              if v is not None:
78                  try:
79                      reduction_template = Template(v)
80                      reduction_vars =
81                      reduction_template.environment.parse(v).find_all(
82                          jinja2.nodes.Name
83                      )
```

```

84     reduction_var_names = {var.name for var in
85     reduction_vars}
86     if "inputs" not in reduction_var_names:
87         raise ValueError(
88             "'resolution_prompt' must contain 'inputs'"
89             variable"
90             )
91     except Exception as e:
92         raise ValueError(
93             f"Invalid Jinja2 template in 'resolution_prompt':"
94             {str(e)}")
95     )
96     return v
97
98     @field_validator("input")
99     def validate_input_schema(cls, v):
100        if v is not None:
101            if "schema" not in v:
102                raise ValueError("Missing 'schema' in 'input' configuration")
103            if not isinstance(v["schema"], dict):
104                raise TypeError(
105                    "'schema' in 'input' configuration must be a dictionary")
106            )
107        return v
108
109
110    @model_validator(mode="after")
111    def validate_output_schema(self, info: ValidationInfo):
112        # Skip validation if we're using from_dataframe accessors
113        if isinstance(info.context, dict) and info.context.get(
114            "_from_df_accessors"
115        ):
116            return self
117
118        if self.output is None:
119            raise ValueError(
120                "Missing required key 'output' in ResolveOperation configuration"
121                )
122
123        if "schema" not in self.output:
124            raise ValueError("Missing 'schema' in 'output' configuration")
125
126        if not isinstance(self.output["schema"], dict):
127            raise TypeError(
128                "'schema' in 'output' configuration must be a dictionary")
129            )
130
131        if not self.output["schema"]:
132            raise ValueError("'schema' in 'output' configuration cannot be empty")
133
134
135        return self
136
137
138    def compare_pair(
139        self,
140        comparison_prompt: str,
141        model: str,
142        
```

```

145     item1: dict,
146     item2: dict,
147     blocking_keys: list[str] = [],
148     timeout_seconds: int = 120,
149     max_retries_per_timeout: int = 2,
150 ) -> tuple[bool, float, str]:
151     """
152         Compares two items using an LLM model to determine if they match.
153
154     Args:
155         comparison_prompt (str): The prompt template for comparison.
156         model (str): The LLM model to use for comparison.
157         item1 (dict): The first item to compare.
158         item2 (dict): The second item to compare.
159
160     Returns:
161         tuple[bool, float, str]: A tuple containing a boolean
162         indicating whether the items match, the cost of the comparison, and the
163         prompt.
164         """
165     if blocking_keys:
166         if all(
167             key in item1
168             and key in item2
169             and str(item1[key]).lower() == str(item2[key]).lower()
170             for key in blocking_keys
171         ):
172             return True, 0, ""
173
174     prompt = strict_render(comparison_prompt, {"input1": item1,
175 "input2": item2})
176     response = self.runner.api.call_llm(
177         model,
178         "compare",
179         [{"role": "user", "content": prompt}],
180         {"is_match": "bool"},
181         timeout_seconds=timeout_seconds,
182         max_retries_per_timeout=max_retries_per_timeout,
183         bypass_cache=self.config.get("bypass_cache",
184         self.bypass_cache),
185
186         litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
187         {}),
188         op_config=self.config,
189     )
190     output = self.runner.api.parse_llm_response(
191         response.response,
192         {"is_match": "bool"},
193     )[0]
194
195     return output["is_match"], response.total_cost, prompt
196
197     def syntax_check(self) -> None:
198         context = {"_from_df_accessors": self.runner._from_df_accessors}
199         super().syntax_check(context)
200
201     def validation_fn(self, response: dict[str, Any]):
202         output = self.runner.api.parse_llm_response(
203             response,
204             schema=self.config["output"]["schema"],
205         )[0]

```

```

206         if self.runner.api.validate_output(self.config, output,
207             self.console):
208             return output, True
209         return output, False
210
211     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
212         """
213             Executes the resolve operation on the provided dataset.
214
215             Args:
216                 input_data (list[dict]): The dataset to resolve.
217
218             Returns:
219                 tuple[list[dict], float]: A tuple containing the resolved
220             results and the total cost of the operation.
221
222             This method performs the following steps:
223             1. Initial blocking based on specified conditions and/or
224             embedding similarity
225                 2. Pairwise comparison of potentially matching entries using LLM
226                 3. Clustering of matched entries
227                 4. Resolution of each cluster into a single entry (if applicable)
228                 5. Result aggregation and validation
229
230             The method also calculates and logs statistics such as
231             comparisons saved by blocking and self-join selectivity.
232
233             """
234         if len(input_data) == 0:
235             return [], 0
236
237         # Initialize observability data for all items at the start
238         if self.config.get("enable_observability", False):
239             observability_key = f"_observability_{self.config['name']}"
240             for item in input_data:
241                 if observability_key not in item:
242                     item[observability_key] = {
243                         "comparison_prompts": [],
244                         "resolution_prompt": None,
245                     }
246
247             blocking_keys = self.config.get("blocking_keys", [])
248             blocking_threshold = self.config.get("blocking_threshold")
249             blocking_conditions = self.config.get("blocking_conditions", [])
250             if self.status:
251                 self.status.stop()
252
253             if not blocking_threshold and not blocking_conditions:
254                 # Prompt the user for confirmation
255                 if not Confirm.ask(
256                     "[yellow]Warning: No blocking keys or conditions
257 specified. "
258                     "This may result in a large number of comparisons. "
259                     "We recommend specifying at least one blocking key or
260                     condition, or using the optimizer to automatically come up with these. "
261                     "Do you want to continue without blocking?[/yellow]",
262                     console=self.runner.console,
263                 ):
264                     raise ValueError("Operation cancelled by user.")
265
266             input_schema = self.config.get("input", {}).get("schema", {})

```

```

267     if not blocking_keys:
268         # Set them to all keys in the input data
269         blocking_keys = list(input_data[0].keys())
270     limit_comparisons = self.config.get("limit_comparisons")
271     total_cost = 0
272
273     def is_match(item1: dict[str, Any], item2: dict[str, Any]) ->
274     bool:
275         return any(
276             eval(condition, {"input1": item1, "input2": item2})
277             for condition in blocking_conditions
278         )
279
280     # Calculate embeddings if blocking_threshold is set
281     embeddings = None
282     if blocking_threshold is not None:
283
284         def get_embeddings_batch(
285             items: list[dict[str, Any]]
286         ) -> list[tuple[list[float], float]]:
287             embedding_model = self.config.get(
288                 "embedding_model", "text-embedding-3-small"
289             )
290             model_input_context_length =
291             model_cost.get(embedding_model, {}).get(
292                 "max_input_tokens", 8192
293             )
294
295             texts = [
296                 " ".join(str(item[key])) for key in blocking_keys if
297                 key in item[
298                     : model_input_context_length * 3
299                 ]
300                 for item in items
301             ]
302
303             response = self.runner.api.gen_embedding(
304                 model=embedding_model, input=texts
305             )
306             return [
307                 (data["embedding"], completion_cost(response))
308                 for data in response["data"]
309             ]
310
311             embeddings = []
312             costs = []
313             with ThreadPoolExecutor(max_workers=self.max_threads) as
314             executor:
315                 for i in range(
316                     0, len(input_data),
317                     self.config.get("embedding_batch_size", 1000)
318                 ):
319                     batch = input_data[
320                         i : i + self.config.get("embedding_batch_size",
321                         1000)
322                     ]
323                     batch_results =
324                     list(executor.map(get_embeddings_batch, [batch]))
325
326                     for result in batch_results:
327                         embeddings.extend([r[0] for r in result])

```

```

328             costs.extend([r[1] for r in result])
329
330         total_cost += sum(costs)
331
332     # Generate all pairs to compare, ensuring no duplicate
333     comparisons
334     def get_unique_comparison_pairs() -> (
335         tuple[list[tuple[int, int]]], dict[tuple[str, ...],
336         list[int]]]
337     ):
338         # Create a mapping of values to their indices
339         value_to_indices: dict[tuple[str, ...], list[int]] = {}
340         for i, item in enumerate(input_data):
341             # Create a hashable key from the blocking keys
342             key = tuple(str(item.get(k, "")) for k in blocking_keys)
343             if key not in value_to_indices:
344                 value_to_indices[key] = []
345             value_to_indices[key].append(i)
346
347         # Generate pairs for comparison, comparing each unique value
348         combination only once
349         comparison_pairs = []
350         keys = list(value_to_indices.keys())
351
352         # First, handle comparisons between different values
353         for i in range(len(keys)):
354             for j in range(i + 1, len(keys)):
355                 # Only need one comparison between different values
356                 idx1 = value_to_indices[keys[i]][0]
357                 idx2 = value_to_indices[keys[j]][0]
358                 if idx1 < idx2: # Maintain ordering to avoid
359                     duplicates
360                     comparison_pairs.append((idx1, idx2))
361
362     return comparison_pairs, value_to_indices
363
364     comparison_pairs, value_to_indices =
365     get_unique_comparison_pairs()
366
367     # Filter pairs based on blocking conditions
368     def meets_blocking_conditions(pair: tuple[int, int]) -> bool:
369         i, j = pair
370         return (
371             is_match(input_data[i], input_data[j]) if
372             blocking_conditions else False
373         )
374
375     blocked_pairs = (
376         list(filter(meets_blocking_conditions, comparison_pairs))
377         if blocking_conditions
378         else comparison_pairs
379     )
380
381     # Apply limit_comparisons to blocked pairs
382     if limit_comparisons is not None and len(blocked_pairs) >
383     limit_comparisons:
384         self.console.log(
385             f"Randomly sampling {limit_comparisons} pairs out of
386             {len(blocked_pairs)} blocked pairs."
387         )
388         blocked_pairs = random.sample(blocked_pairs,

```

```

389     limit_comparisons)
390
391     # Initialize clusters with all indices
392     clusters = [{i} for i in range(len(input_data))]
393     cluster_map = {i: i for i in range(len(input_data))}
394
395     # If there are remaining comparisons, fill with highest cosine
396     similarities
397     remaining_comparisons = (
398         limit_comparisons - len(blocked_pairs)
399         if limit_comparisons is not None
400         else float("inf")
401     )
402     if remaining_comparisons > 0 and blocking_threshold is not None:
403         # Compute cosine similarity for all pairs efficiently
404         from sklearn.metrics.pairwise import cosine_similarity
405
406         similarity_matrix = cosine_similarity(embeddings)
407
408         cosine_pairs = []
409         for i, j in comparison_pairs:
410             if (i, j) not in blocked_pairs and find_cluster(
411                 i, cluster_map
412             ) != find_cluster(j, cluster_map):
413                 similarity = similarity_matrix[i, j]
414                 if similarity >= blocking_threshold:
415                     cosine_pairs.append((i, j, similarity))
416
417         if remaining_comparisons != float("inf"):
418             cosine_pairs.sort(key=lambda x: x[2], reverse=True)
419             additional_pairs = [
420                 (i, j) for i, j, _ in cosine_pairs[:int(remaining_comparisons)]
421             ]
422             blocked_pairs.extend(additional_pairs)
423         else:
424             blocked_pairs.extend((i, j) for i, j, _ in cosine_pairs)
425
426         # Modified merge_clusters to handle all indices with the same
427         value
428
429
430     def merge_clusters(item1: int, item2: int) -> None:
431         root1, root2 = find_cluster(item1, cluster_map),
432         find_cluster(
433             item2, cluster_map
434         )
435         if root1 != root2:
436             if len(clusters[root1]) < len(clusters[root2]):
437                 root1, root2 = root2, root1
438                 clusters[root1] |= clusters[root2]
439                 cluster_map[root2] = root1
440                 clusters[root2] = set()
441
442             # Also merge all other indices that share the same values
443             key1 = tuple(str(input_data[item1].get(k, "")) for k in
444             blocking_keys)
445             key2 = tuple(str(input_data[item2].get(k, "")) for k in
446             blocking_keys)
447
448             # Merge all indices with the same values
449             for idx in value_to_indices.get(key1, []):

```

```

450             if idx != item1:
451                 root_idx = find_cluster(idx, cluster_map)
452                 if root_idx != root1:
453                     clusters[root1] |= clusters[root_idx]
454                     cluster_map[root_idx] = root1
455                     clusters[root_idx] = set()
456
457             for idx in value_to_indices.get(key2, []):
458                 if idx != item2:
459                     root_idx = find_cluster(idx, cluster_map)
460                     if root_idx != root1:
461                         clusters[root1] |= clusters[root_idx]
462                         cluster_map[root_idx] = root1
463                         clusters[root_idx] = set()
464
465             # Calculate and print statistics
466             total_possible_comparisons = len(input_data) * (len(input_data) -
467             1) // 2
468             comparisons_made = len(blocked_pairs)
469             comparisons_saved = total_possible_comparisons - comparisons_made
470             self.console.log(
471                 f"[green]Comparisons saved by blocking: {comparisons_saved} "
472                 f"({{comparisons_saved / total_possible_comparisons) * "
473                 "100:.2f}}%)[/green]"
474             )
475             self.console.log(
476                 f"[blue]Number of pairs to compare: {len(blocked_pairs)}"
477                 "[/blue]"
478             )
479
480             # Compute an auto-batch size based on the number of comparisons
481             def auto_batch() -> int:
482                 # Maximum batch size limit for 4o-mini model
483                 M = 500
484
485                 n = len(input_data)
486                 m = len(blocked_pairs)
487
488                 # https://www.wolframalpha.com/input/?i=k%28k-
489                 1%29%2F2+%2B%28n-k%29%28k-1%29+3D+m%2C+solve+for+k
490                 # Two possible solutions for k:
491                 # k = -1/2 sqrt((1 - 2n)^2 - 8m) + n + 1/2
492                 # k = 1/2 (sqrt((1 - 2n)^2 - 8m) + 2n + 1)
493
494                 discriminant = (1 - 2 * n) ** 2 - 8 * m
495                 sqrt_discriminant = discriminant**0.5
496
497                 k1 = -0.5 * sqrt_discriminant + n + 0.5
498                 k2 = 0.5 * (sqrt_discriminant + 2 * n + 1)
499
500                 # Take the maximum viable solution
501                 k = max(k1, k2)
502                 return M if k < 0 else min(int(k), M)
503
504             # Compare pairs and update clusters in real-time
505             batch_size = self.config.get("compare_batch_size", auto_batch())
506             self.console.log(f"Using compare batch size: {batch_size}")
507             pair_costs = 0
508
509             pbar = RichLoopBar(
510                 range(0, len(blocked_pairs), batch_size),

```

```

511         desc=f"Processing batches of {batch_size} LLM comparisons",
512         console=self.console,
513     )
514     last_processed = 0
515     for i in pbar:
516         batch_end = last_processed + batch_size
517         batch = blocked_pairs[last_processed:batch_end]
518         # Filter pairs for the initial batch
519         better_batch = [
520             pair
521             for pair in batch
522             if find_cluster(pair[0], cluster_map) == pair[0]
523             and find_cluster(pair[1], cluster_map) == pair[1]
524         ]
525
526         # Expand better_batch if it doesn't reach batch_size
527         while len(better_batch) < batch_size and batch_end <
528             len(blocked_pairs):
529                 # Move batch_end forward by batch_size to get more pairs
530                 next_end = batch_end + batch_size
531                 next_batch = blocked_pairs[batch_end:next_end]
532
533                 better_batch.extend(
534                     pair
535                     for pair in next_batch
536                     if find_cluster(pair[0], cluster_map) == pair[0]
537                     and find_cluster(pair[1], cluster_map) == pair[1]
538                 )
539
540         # Update batch_end to prevent overlapping in the next
541     loop
542         batch_end = next_end
543         better_batch = better_batch[:batch_size]
544         last_processed = batch_end
545         with ThreadPoolExecutor(max_workers=self.max_threads) as
546             executor:
547                 future_to_pair = {
548                     executor.submit(
549                         self.compare_pair,
550                         self.config["comparison_prompt"],
551                         self.config.get("comparison_model",
552                         self.default_model),
553                         input_data[pair[0]],
554                         input_data[pair[1]],
555                         blocking_keys,
556                         timeout_seconds=self.config.get("timeout", 120),
557                         max_retries_per_timeout=self.config.get(
558                             "max_retries_per_timeout", 2
559                         ),
560                         ): pair
561                         for pair in better_batch
562                 }
563
564                 for future in as_completed(future_to_pair):
565                     pair = future_to_pair[future]
566                     is_match_result, cost, prompt = future.result()
567                     pair_costs += cost
568                     if is_match_result:
569                         merge_clusters(pair[0], pair[1])
570
571                     if self.config.get("enable_observability", False):

```

```

572         observability_key =
573     f"_observability_{self.config['name']}\""
574     for idx in (pair[0], pair[1]):
575         if observability_key not in input_data[idx]:
576             input_data[idx][observability_key] = {
577                 "comparison_prompts": [],
578                 "resolution_prompt": None,
579             }
580         input_data[idx][observability_key][
581             "comparison_prompts"
582         ].append(prompt)
583
584     total_cost += pair_costs
585
586     # Collect final clusters
587     final_clusters = [cluster for cluster in clusters if cluster]
588
589     # Process each cluster
590     results = []
591
592     def process_cluster(cluster):
593         if len(cluster) > 1:
594             cluster_items = [input_data[i] for i in cluster]
595             if input_schema:
596                 cluster_items = [
597                     {k: item[k] for k in input_schema.keys() if k in
598                      item}
599                     for item in cluster_items
600                 ]
601
602             resolution_prompt = strict_render(
603                 self.config["resolution_prompt"], {"inputs":
604                     cluster_items})
605
606             reduction_response = self.runner.api.call_llm(
607                 self.config.get("resolution_model",
608                 self.default_model),
609                 "reduce",
610                 [{"role": "user", "content": resolution_prompt}],
611                 self.config["output"]["schema"],
612                 timeout_seconds=self.config.get("timeout", 120),
613                 max_retries_per_timeout=self.config.get(
614                     "max_retries_per_timeout", 2
615                 ),
616                 bypass_cache=self.config.get("bypass_cache",
617                 self.bypass_cache),
618                 validation_config=(
619                     {
620                         "val_rule": self.config.get("validate", []),
621                         "validation_fn": self.validation_fn,
622                     }
623                     if self.config.get("validate", None)
624                     else None
625                 ),
626                 litellm_completion_kwargs=self.config.get(
627                     "litellm_completion_kwargs", {})
628                 ),
629                 op_config=self.config,
630             )
631             reduction_cost = reduction_response.total_cost
632

```

```

633         if self.config.get("enable_observability", False):
634             for item in [input_data[i] for i in cluster]:
635                 observability_key =
636 f"_{observability}_{self.config['name']}_{}"
637                 if observability_key not in item:
638                     item[observability_key] = {
639                         "comparison_prompts": [],
640                         "resolution_prompt": None,
641                     }
642                     item[observability_key]["resolution_prompt"] =
643             resolution_prompt
644
645             if reduction_response.validated:
646                 reduction_output =
647 self.runner.api.parse_llm_response(
648                 reduction_response.response,
649                 self.config["output"]["schema"],
650                 manually_fix_errors=self.manually_fix_errors,
651             )[0]
652
653             # If the output is overwriting an existing key, we
654             want to save the kv pairs
655             keys_in_output = [
656                 k
657                 for k in set(reduction_output.keys())
658                 if k in cluster_items[0].keys()
659             ]
660
661             return (
662                 [
663                     {
664                         **item,
665
666                         f"_{kv_pairs_preresolve}_{self.config['name']}": {
667                             k: item[k] for k in keys_in_output
668                         },
669                         **{
670                             k: reduction_output[k]
671                             for k in self.config["output"]
672                         ["schema"]
673                         },
674                         ],
675                         for item in [input_data[i] for i in cluster]
676                     ],
677                         reduction_cost,
678                     )
679             return [], reduction_cost
680         else:
681             # Set the output schema to be the keys found in the
682             compare_prompt
683             compare_prompt_keys = extract_jinja_variables(
684                 self.config["comparison_prompt"]
685             )
686             # Get the set of keys in the compare_prompt
687             compare_prompt_keys = set(
688                 [
689                     k.replace("input1.", "")
690                     for k in compare_prompt_keys
691                     if "input1" in k
692                 ]
693             )

```

```

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
709                         best_match = compare_key
710                         key_mapping[output_key] = best_match
711
712             # Create the result dictionary using the key mapping
713             result = input_data[list(cluster)[0]].copy()
714             result[f"_kv_pairs_preresolve_{self.config['name']}"] = {
715                 ok: result[ck] for ok, ck in key_mapping.items() if
716                 ck in result
717             }
718             for output_key, compare_key in key_mapping.items():
719                 if compare_key in input_data[list(cluster)[0]]:
720                     result[output_key] = input_data[list(cluster)[0]][
721                         compare_key]
722                 elif output_key in input_data[list(cluster)[0]]:
723                     result[output_key] = input_data[list(cluster)[0]][
724                         output_key]
725                 else:
726                     result[output_key] = None # or some default
727                     value
728
729             return [result], 0
730
731             # Calculate the number of records before and clusters after
732             num_records_before = len(input_data)
733             num_clusters_after = len(final_clusters)
734             self.console.log(f"Number of keys before resolution:
735 {num_records_before}")
736             self.console.log(
737                 f"Number of distinct keys after resolution:
738 {num_clusters_after}"
739             )
740
741             # If no resolution prompt is provided, we can skip the resolution
742             phase
743             # And simply select the most common value for each key
744             if not self.config.get("resolution_prompt", None):
745                 for cluster in final_clusters:
746                     if len(cluster) > 1:
747                         for key in self.config["output"]["keys"]:
748                             most_common_value = max(
749                                 set(input_data[i][key] for i in cluster),
750                                 key=lambda x: sum(
751                                     1 for i in cluster if input_data[i][key]
752                                     == x
753                                 ),
754                             )
755

```

```

                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.	<i>required</i>
model	str	The LLM model to use for comparison.	<i>required</i>

Name	Type	Description	Default
item1	dict	The first item to compare.	<i>required</i>
item2	dict	The second item to compare.	<i>required</i>

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`

```

126     def compare_pair(
127         self,
128         comparison_prompt: str,
129         model: str,
130         item1: dict,
131         item2: dict,
132         blocking_keys: list[str] = [],
133         timeout_seconds: int = 120,
134         max_retries_per_timeout: int = 2,
135     ) -> tuple[bool, float, str]:
136         """
137             Compares two items using an LLM model to determine if they match.
138
139             Args:
140                 comparison_prompt (str): The prompt template for comparison.
141                 model (str): The LLM model to use for comparison.
142                 item1 (dict): The first item to compare.
143                 item2 (dict): The second item to compare.
144
145             Returns:
146                 tuple[bool, float, str]: A tuple containing a boolean indicating
147                 whether the items match, the cost of the comparison, and the prompt.
148                 """
149             if blocking_keys:
150                 if all(
151                     key in item1
152                     and key in item2
153                     and str(item1[key]).lower() == str(item2[key]).lower()
154                     for key in blocking_keys
155                 ):
156                     return True, 0, ""
157
158             prompt = strict_render(comparison_prompt, {"input1": item1, "input2": item2})
159             response = self.runner.api.call_llm(
160                 model,
161                 "compare",
162                 [{"role": "user", "content": prompt}],
163                 {"is_match": "bool"},
164                 timeout_seconds=timeout_seconds,
165                 max_retries_per_timeout=max_retries_per_timeout,
166                 bypass_cache=self.config.get("bypass_cache", self.bypass_cache),
167
168             litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
169             {}),
170             op_config=self.config,
171         )
172             output = self.runner.api.parse_llm_response(
173                 response.response,
174                 {"is_match": "bool"}, [0]
175
176             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.	<i>required</i>

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`

```
189     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
190         """
191             Executes the resolve operation on the provided dataset.
192
193             Args:
194                 input_data (list[dict]): The dataset to resolve.
195
196             Returns:
197                 tuple[list[dict], float]: A tuple containing the resolved results
198                 and the total cost of the operation.
199
200             This method performs the following steps:
201                 1. Initial blocking based on specified conditions and/or embedding
202                     similarity
203                     2. Pairwise comparison of potentially matching entries using LLM
204                     3. Clustering of matched entries
205                     4. Resolution of each cluster into a single entry (if applicable)
206                     5. Result aggregation and validation
207
208             The method also calculates and logs statistics such as comparisons
209             saved by blocking and self-join selectivity.
210             """
211     if len(input_data) == 0:
212         return [], 0
213
214     # Initialize observability data for all items at the start
215     if self.config.get("enable_observability", False):
216         observability_key = f"_observability_{self.config['name']}"
217         for item in input_data:
218             if observability_key not in item:
219                 item[observability_key] = {
220                     "comparison_prompts": [],
221                     "resolution_prompt": None,
222                 }
223
224     blocking_keys = self.config.get("blocking_keys", [])
225     blocking_threshold = self.config.get("blocking_threshold")
226     blocking_conditions = self.config.get("blocking_conditions", [])
227     if self.status:
228         self.status.stop()
229
230     if not blocking_threshold and not blocking_conditions:
231         # Prompt the user for confirmation
232         if not Confirm.ask(
233             "[yellow]Warning: No blocking keys or conditions specified. "
234             "This may result in a large number of comparisons. "
235             "We recommend specifying at least one blocking key or
236             condition, or using the optimizer to automatically come up with these. "
237             "Do you want to continue without blocking?[/yellow]",
238             console=self.runner.console,
239         ):
240             raise ValueError("Operation cancelled by user.")
241
242     input_schema = self.config.get("input", {}).get("schema", {})
243     if not blocking_keys:
244         # Set them to all keys in the input data
245         blocking_keys = list(input_data[0].keys())
```

```

246     limit_comparisons = self.config.get("limit_comparisons")
247     total_cost = 0
248
249     def is_match(item1: dict[str, Any], item2: dict[str, Any]) -> bool:
250         return any(
251             eval(condition, {"input1": item1, "input2": item2})
252             for condition in blocking_conditions
253         )
254
255     # Calculate embeddings if blocking_threshold is set
256     embeddings = None
257     if blocking_threshold is not None:
258
259         def get_embeddings_batch(
260             items: list[dict[str, Any]]
261         ) -> list[tuple[list[float], float]]:
262             embedding_model = self.config.get(
263                 "embedding_model", "text-embedding-3-small"
264             )
265             model_input_context_length = model_cost.get(embedding_model,
266             {}).get(
267                 "max_input_tokens", 8192
268             )
269
270             texts = [
271                 " ".join(str(item[key])) for key in blocking_keys if key
272             in item[
273                 : model_input_context_length * 3
274             ]
275             for item in items
276             ]
277
278             response = self.runner.api.gen_embedding(
279                 model=embedding_model, input=texts
280             )
281             return [
282                 (data["embedding"], completion_cost(response))
283                 for data in response["data"]
284             ]
285
286             embeddings = []
287             costs = []
288             with ThreadPoolExecutor(max_workers=self.max_threads) as
289             executor:
290                 for i in range(
291                     0, len(input_data),
292                     self.config.get("embedding_batch_size", 1000)
293                 ):
294                     batch = input_data[
295                         i : i + self.config.get("embedding_batch_size", 1000)
296                     ]
297                     batch_results = list(executor.map(get_embeddings_batch,
298                     [batch]))
299
300                     for result in batch_results:
301                         embeddings.extend([r[0] for r in result])
302                         costs.extend([r[1] for r in result])
303
304                     total_cost += sum(costs)
305
306             # Generate all pairs to compare, ensuring no duplicate comparisons

```

```

307     def get_unique_comparison_pairs() -> (
308         tuple[list[tuple[int, int]]], dict[tuple[str, ...], list[int]]
309     ):
310         # Create a mapping of values to their indices
311         value_to_indices: dict[tuple[str, ...], list[int]] = {}
312         for i, item in enumerate(input_data):
313             # Create a hashable key from the blocking keys
314             key = tuple(str(item.get(k, "")) for k in blocking_keys)
315             if key not in value_to_indices:
316                 value_to_indices[key] = []
317             value_to_indices[key].append(i)
318
319         # Generate pairs for comparison, comparing each unique value
320         # combination only once
321         comparison_pairs = []
322         keys = list(value_to_indices.keys())
323
324         # First, handle comparisons between different values
325         for i in range(len(keys)):
326             for j in range(i + 1, len(keys)):
327                 # Only need one comparison between different values
328                 idx1 = value_to_indices[keys[i]][0]
329                 idx2 = value_to_indices[keys[j]][0]
330                 if idx1 < idx2: # Maintain ordering to avoid duplicates
331                     comparison_pairs.append((idx1, idx2))
332
333         return comparison_pairs, value_to_indices
334
335     comparison_pairs, value_to_indices = get_unique_comparison_pairs()
336
337     # Filter pairs based on blocking conditions
338     def meets_blocking_conditions(pair: tuple[int, int]) -> bool:
339         i, j = pair
340         return (
341             is_match(input_data[i], input_data[j]) if blocking_conditions
342         else False
343         )
344
345     blocked_pairs = (
346         list(filter(meets_blocking_conditions, comparison_pairs))
347         if blocking_conditions
348         else comparison_pairs
349     )
350
351     # Apply limit_comparisons to blocked pairs
352     if limit_comparisons is not None and len(blocked_pairs) >
353     limit_comparisons:
354         self.console.log(
355             f"Randomly sampling {limit_comparisons} pairs out of
356             {len(blocked_pairs)} blocked pairs."
357         )
358         blocked_pairs = random.sample(blocked_pairs, limit_comparisons)
359
360     # Initialize clusters with all indices
361     clusters = [{i} for i in range(len(input_data))]
362     cluster_map = {i: i for i in range(len(input_data))}
363
364     # If there are remaining comparisons, fill with highest cosine
365     # similarities
366     remaining_comparisons = (
367         limit_comparisons - len(blocked_pairs)

```

```

368     if limit_comparisons is not None
369         else float("inf")
370     )
371     if remaining_comparisons > 0 and blocking_threshold is not None:
372         # Compute cosine similarity for all pairs efficiently
373         from sklearn.metrics.pairwise import cosine_similarity
374
375         similarity_matrix = cosine_similarity(embeddings)
376
377         cosine_pairs = []
378         for i, j in comparison_pairs:
379             if (i, j) not in blocked_pairs and find_cluster(
380                 i, cluster_map
381             ) != find_cluster(j, cluster_map):
382                 similarity = similarity_matrix[i, j]
383                 if similarity >= blocking_threshold:
384                     cosine_pairs.append((i, j, similarity))
385
386         if remaining_comparisons != float("inf"):
387             cosine_pairs.sort(key=lambda x: x[2], reverse=True)
388             additional_pairs = [
389                 (i, j) for i, j, _ in cosine_pairs[:int(remaining_comparisons)]
390             ]
391             blocked_pairs.extend(additional_pairs)
392         else:
393             blocked_pairs.extend((i, j) for i, j, _ in cosine_pairs)
394
395         # Modified merge_clusters to handle all indices with the same value
396
397     def merge_clusters(item1: int, item2: int) -> None:
398         root1, root2 = find_cluster(item1, cluster_map), find_cluster(
399             item2, cluster_map
400         )
401         if root1 != root2:
402             if len(clusters[root1]) < len(clusters[root2]):
403                 root1, root2 = root2, root1
404                 clusters[root1] |= clusters[root2]
405                 cluster_map[root2] = root1
406                 clusters[root2] = set()
407
408             # Also merge all other indices that share the same values
409             key1 = tuple(str(input_data[item1].get(k, "")) for k in
410             blocking_keys)
411             key2 = tuple(str(input_data[item2].get(k, "")) for k in
412             blocking_keys)
413
414             # Merge all indices with the same values
415             for idx in value_to_indices.get(key1, []):
416                 if idx != item1:
417                     root_idx = find_cluster(idx, cluster_map)
418                     if root_idx != root1:
419                         clusters[root1] |= clusters[root_idx]
420                         cluster_map[root_idx] = root1
421                         clusters[root_idx] = set()
422
423             for idx in value_to_indices.get(key2, []):
424                 if idx != item2:
425                     root_idx = find_cluster(idx, cluster_map)
426                     if root_idx != root1:
427                         clusters[root1] |= clusters[root_idx]

```

```

429             cluster_map[root_idx] = root1
430             clusters[root_idx] = set()
431
432     # Calculate and print statistics
433     total_possible_comparisons = len(input_data) * (len(input_data) - 1)
434     // 2
435     comparisons_made = len(blocked_pairs)
436     comparisons_saved = total_possible_comparisons - comparisons_made
437     self.console.log(
438         f"[green]Comparisons saved by blocking: {comparisons_saved} "
439         f"({{comparisons_saved / total_possible_comparisons} * 100:.2f}%)"
440     [/green]")
441     )
442     self.console.log(
443         f"[blue]Number of pairs to compare: {len(blocked_pairs)}[/blue]"
444     )
445
446     # Compute an auto-batch size based on the number of comparisons
447     def auto_batch() -> int:
448         # Maximum batch size limit for 4o-mini model
449         M = 500
450
451         n = len(input_data)
452         m = len(blocked_pairs)
453
454         # https://www.wolframalpha.com/input/?i=k%28k-1%29%2F2+%2B+%28n-
455         k%29%28k-1%29+3D+m%2C+solve+for+k
456         # Two possible solutions for k:
457         # k = -1/2 sqrt((1 - 2n)^2 - 8m) + n + 1/2
458         # k = 1/2 (sqrt((1 - 2n)^2 - 8m) + 2n + 1)
459
460         discriminant = (1 - 2 * n) ** 2 - 8 * m
461         sqrt_discriminant = discriminant**0.5
462
463         k1 = -0.5 * sqrt_discriminant + n + 0.5
464         k2 = 0.5 * (sqrt_discriminant + 2 * n + 1)
465
466         # Take the maximum viable solution
467         k = max(k1, k2)
468         return M if k < 0 else min(int(k), M)
469
470     # Compare pairs and update clusters in real-time
471     batch_size = self.config.get("compare_batch_size", auto_batch())
472     self.console.log(f"Using compare batch size: {batch_size}")
473     pair_costs = 0
474
475     pbar = RichLoopBar(
476         range(0, len(blocked_pairs), batch_size),
477         desc=f"Processing batches of {batch_size} LLM comparisons",
478         console=self.console,
479     )
480     last_processed = 0
481     for i in pbar:
482         batch_end = last_processed + batch_size
483         batch = blocked_pairs[last_processed:batch_end]
484         # Filter pairs for the initial batch
485         better_batch = [
486             pair
487             for pair in batch
488             if find_cluster(pair[0], cluster_map) == pair[0]
489             and find_cluster(pair[1], cluster_map) == pair[1]

```

```

490     ]
491
492     # Expand better_batch if it doesn't reach batch_size
493     while len(better_batch) < batch_size and batch_end <
494     len(blocked_pairs):
495         # Move batch_end forward by batch_size to get more pairs
496         next_end = batch_end + batch_size
497         next_batch = blocked_pairs[batch_end:next_end]
498
499         better_batch.extend(
500             pair
501                 for pair in next_batch
502                     if find_cluster(pair[0], cluster_map) == pair[0]
503                     and find_cluster(pair[1], cluster_map) == pair[1]
504             )
505
506         # Update batch_end to prevent overlapping in the next loop
507         batch_end = next_end
508         better_batch = better_batch[:batch_size]
509         last_processed = batch_end
510         with ThreadPoolExecutor(max_workers=self.max_threads) as
511     executor:
512             future_to_pair = {
513                 executor.submit(
514                     self.compare_pair,
515                     self.config["comparison_prompt"],
516                     self.config.get("comparison_model",
517                     self.default_model),
518                     input_data[pair[0]],
519                     input_data[pair[1]],
520                     blocking_keys,
521                     timeout_seconds=self.config.get("timeout", 120),
522                     max_retries_per_timeout=self.config.get(
523                         "max_retries_per_timeout", 2
524                     ),
525                     ): pair
526                     for pair in better_batch
527             }
528
529             for future in as_completed(future_to_pair):
530                 pair = future_to_pair[future]
531                 is_match_result, cost, prompt = future.result()
532                 pair_costs += cost
533                 if is_match_result:
534                     merge_clusters(pair[0], pair[1])
535
536                     if self.config.get("enable_observability", False):
537                         observability_key =
538                         f"_observability_{self.config['name']}@"
539                         for idx in (pair[0], pair[1]):
540                             if observability_key not in input_data[idx]:
541                                 input_data[idx][observability_key] = {
542                                     "comparison_prompts": [],
543                                     "resolution_prompt": None,
544                                 }
545                                 input_data[idx][observability_key][
546                                     "comparison_prompts"
547                                 ].append(prompt)
548
549             total_cost += pair_costs
550

```

```

551     # Collect final clusters
552     final_clusters = [cluster for cluster in clusters if cluster]
553
554     # Process each cluster
555     results = []
556
557     def process_cluster(cluster):
558         if len(cluster) > 1:
559             cluster_items = [input_data[i] for i in cluster]
560             if input_schema:
561                 cluster_items = [
562                     {k: item[k] for k in input_schema.keys() if k in
563                      item}
564                     for item in cluster_items
565                 ]
566
567             resolution_prompt = strict_render(
568                 self.config["resolution_prompt"], {"inputs":
569                 cluster_items})
570         )
571         reduction_response = self.runner.api.call_llm(
572             self.config.get("resolution_model", self.default_model),
573             "reduce",
574             [{"role": "user", "content": resolution_prompt}],
575             self.config["output"]["schema"],
576             timeout_seconds=self.config.get("timeout", 120),
577             max_retries_per_timeout=self.config.get(
578                 "max_retries_per_timeout", 2
579             ),
580             bypass_cache=self.config.get("bypass_cache",
581             self.bypass_cache),
582             validation_config=(
583                 {
584                     "val_rule": self.config.get("validate", []),
585                     "validation_fn": self.validation_fn,
586                 }
587                 if self.config.get("validate", None)
588                 else None
589             ),
590             litellm_completion_kwargs=self.config.get(
591                 "litellm_completion_kwargs", {}
592             ),
593             op_config=self.config,
594         )
595         reduction_cost = reduction_response.total_cost
596
597         if self.config.get("enable_observability", False):
598             for item in [input_data[i] for i in cluster]:
599                 observability_key =
600 f"observability_{self.config['name']}\""
601                 if observability_key not in item:
602                     item[observability_key] = {
603                         "comparison_prompts": [],
604                         "resolution_prompt": None,
605                     }
606                     item[observability_key]["resolution_prompt"] =
607             resolution_prompt
608
609             if reduction_response.validated:
610                 reduction_output = self.runner.api.parse_llm_response(
611                     reduction_response.response,

```

```

612             self.config["output"]["schema"],
613             manually_fix_errors=self.manually_fix_errors,
614         )[0]
615
616         # If the output is overwriting an existing key, we want
617         # to save the kv pairs
618         keys_in_output = [
619             k
620             for k in set(reduction_output.keys())
621             if k in cluster_items[0].keys()
622         ]
623
624         return (
625             [
626                 {
627                     **item,
628
629             f"_kv_pairs_preresolve_{self.config['name']}": {
630                 k: item[k] for k in keys_in_output
631             },
632             **{
633                 k: reduction_output[k]
634                 for k in self.config["output"]["schema"]
635             },
636             }
637             for item in [input_data[i] for i in cluster]
638         ],
639         reduction_cost,
640     )
641     return [], reduction_cost
642 else:
643     # Set the output schema to be the keys found in the
644     compare_prompt
645     compare_prompt_keys = extract_jinja_variables(
646         self.config["comparison_prompt"]
647     )
648     # Get the set of keys in the compare_prompt
649     compare_prompt_keys = set(
650         [
651             k.replace("input1.", "")
652             for k in compare_prompt_keys
653             if "input1" in k
654         ]
655     )
656
657     # For each key in the output schema, find the most similar
658     # key in the compare_prompt
659     output_keys = set(self.config["output"]["schema"].keys())
660     key_mapping = {}
661     for output_key in output_keys:
662         best_match = None
663         best_score = 0
664         for compare_key in compare_prompt_keys:
665             score = sum(
666                 c1 == c2 for c1, c2 in zip(output_key,
667                 compare_key)
668             ) / max(len(output_key), len(compare_key))
669             if score > best_score:
670                 best_score = score
671                 best_match = compare_key
672             key_mapping[output_key] = best_match

```

```

673         # Create the result dictionary using the key mapping
674         result = input_data[list(cluster)[0]].copy()
675         result[f"_kv_pairs_preresolve_{self.config['name']}"] = {
676             ok: result[ck] for ok, ck in key_mapping.items() if ck in
677             result
678         }
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(
709                         set(input_data[i][key] for i in cluster),
710                         key=lambda x: sum(
711                             1 for i in cluster if input_data[i][key] == x
712                         ),
713                     )
714                     for i in cluster:
715                         input_data[i][key] = most_common_value
716
717             results = input_data
718         else:
719             with ThreadPoolExecutor(max_workers=self.max_threads) as
720             executor:
721                 futures = [
722                     executor.submit(process_cluster, cluster)
723                     for cluster in final_clusters
724                 ]
725                 for future in rich_as_completed(
726                     futures,
727                     total=len(futures),
728                     desc="Determining resolved key for each group of
729                     equivalent keys",
730                     console=self.console,
731                 ):
732                     cluster_results, cluster_cost = future.result()
733                     results.extend(cluster_results)
734                     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`

```
34     class ReduceOperation(BaseOperation):
35         """
36             A class that implements a reduce operation on input data using
37             language models.
38
39             This class extends BaseOperation to provide functionality for
40             reducing grouped data
41             using various strategies including batch reduce, incremental reduce,
42             and parallel fold and merge.
43             """
44
45     class schema(BaseOperation.schema):
46         type: str = "reduce"
47         reduce_key: str | list[str]
48         output: dict[str, Any]
49         prompt: str
50         optimize: bool | None = None
51         synthesize_resolve: bool | None = None
52         model: str | None = None
53         input: dict[str, Any] | None = None
54         pass_through: bool | None = None
55         associative: bool | None = None
56         fold_prompt: str | None = None
57         fold_batch_size: int | None = Field(None, gt=0)
58         merge_prompt: str | None = None
59         merge_batch_size: int | None = Field(None, gt=0)
60         value_sampling: dict[str, Any] | None = None
61         verbose: bool | None = None
62         timeout: int | None = None
63         litellm_completion_kwargs: dict[str, Any] =
64             Field(default_factory=dict)
65         enable_observability: bool = False
66
67         @field_validator("prompt")
68         def validate_prompt(cls, v):
69             if v is not None:
70                 try:
71                     template = Template(v)
72                     template_vars =
73                     template.environment.parse(v).find_all(
74                         jinja2.nodes.Name
75                     )
76                     template_var_names = {var.name for var in
77                     template_vars}
78                     if "inputs" not in template_var_names:
79                         raise ValueError(
80                             "Prompt template must include the 'inputs'"
81                             "variable"
82                         )
83                 except Exception as e:
84                     raise ValueError(f"Invalid Jinja2 template in
85 'prompt': {str(e)}")
86             return v
87
88         @field_validator("fold_prompt")
89         def validate_fold_prompt(cls, v):
90             if v is not None:
```

```

91         try:
92             fold_template = Template(v)
93             fold_template_vars =
94             fold_template.environment.parse(v).find_all(
95                 jinja2.nodes.Name
96             )
97             fold_template_var_names = {var.name for var in
98             fold_template_vars}
99             required_vars = {"inputs", "output"}
100            if not
101            required_vars.issubset(fold_template_var_names):
102                raise ValueError(
103                    f"Fold template must include variables:
104 {required_vars}. Current template includes: {fold_template_var_names}"
105                )
106            except Exception as e:
107                raise ValueError(
108                    f"Invalid Jinja2 template in 'fold_prompt':
109 {str(e)}"
110                )
111            return v
112
113     @field_validator("merge_prompt")
114     def validate_merge_prompt(cls, v):
115         if v is not None:
116             try:
117                 merge_template = Template(v)
118                 merge_template_vars =
119                 merge_template.environment.parse(v).find_all(
120                     jinja2.nodes.Name
121                 )
122                 merge_template_var_names = {var.name for var in
123                 merge_template_vars}
124                 if "outputs" not in merge_template_var_names:
125                     raise ValueError(
126                         "Merge template must include the 'outputs'
127 variable"
128                     )
129                 except Exception as e:
130                     raise ValueError(
131                         f"Invalid Jinja2 template in 'merge_prompt':
132 {str(e)}"
133                     )
134             return v
135
136     @field_validator("value_sampling")
137     def validate_value_sampling(cls, v):
138         if v is not None:
139             if v["enabled"]:
140                 if v["method"] not in ["random", "first_n",
141 "cluster", "sem_sim"]:
142                     raise ValueError(
143                         "Invalid 'method'. Must be 'random',
144 'first_n', 'cluster', or 'sem_sim'"
145                     )
146
147                 if v["method"] == "embedding":
148                     if "embedding_model" not in v:
149                         raise ValueError(
150                             "'embedding_model' is required when using
151 embedding-based sampling"

```

```

152                     )
153             if "embedding_keys" not in v:
154                 raise ValueError(
155                     "'embedding_keys' is required when using
156             embedding-based sampling"
157                     )
158         return v
159
160     @model_validator(mode="after")
161     def validate_complex_requirements(self):
162         # Check dependencies between merge_prompt and fold_prompt
163         if self.merge_prompt and not self.fold_prompt:
164             raise ValueError(
165                 "'fold_prompt' is required when 'merge_prompt' is
166             specified"
167                     )
168
169         # Check batch size requirements
170         if self.fold_prompt and not self.fold_batch_size:
171             raise ValueError(
172                 "'fold_batch_size' is required when 'fold_prompt' is
173             specified"
174                     )
175         if self.merge_prompt and not self.merge_batch_size:
176             raise ValueError(
177                 "'merge_batch_size' is required when 'merge_prompt'
178             is specified"
179                     )
180
181         return self
182
183     def __init__(self, *args, **kwargs):
184         """
185             Initialize the ReduceOperation.
186
187             Args:
188                 *args: Variable length argument list.
189                 **kwargs: Arbitrary keyword arguments.
190             """
191         super().__init__(*args, **kwargs)
192         self.min_samples = 5
193         self.max_samples = 1000
194         self.fold_times = deque(maxlen=self.max_samples)
195         self.merge_times = deque(maxlen=self.max_samples)
196         self.lock = Lock()
197         self.config["reduce_key"] = (
198             [self.config["reduce_key"]]
199             if isinstance(self.config["reduce_key"], str)
200             else self.config["reduce_key"]
201         )
202         self.intermediates = {}
203         self.lineage_keys = self.config.get("output", {}).get("lineage",
204             [])
205
206     def execute(self, input_data: list[dict]) -> tuple[list[dict],
207         float]:
208         """
209             Execute the reduce operation on the provided input data.
210
211             This method sorts and groups the input data by the reduce key(s),
212             then processes each group

```

```

213         using either parallel fold and merge, incremental reduce, or
214     batch reduce strategies.
215
216     Args:
217         input_data (list[dict]): The input data to process.
218
219     Returns:
220         tuple[list[dict], float]: A tuple containing the processed
221     results and the total cost of the operation.
222         """
223         if self.config.get("gleaning", {}).get("validation_prompt",
224     None):
225             self.console.log(
226                 f"Using gleaning with validation prompt:
227 {self.config.get('gleaning', {}).get('validation_prompt', '')}"
228             )
229
230         reduce_keys = self.config["reduce_key"]
231         if isinstance(reduce_keys, str):
232             reduce_keys = [reduce_keys]
233         input_schema = self.config.get("input", {}).get("schema", {})
234
235         if self.status:
236             self.status.stop()
237
238         # Check if we need to group everything into one group
239         if reduce_keys == ["_all"] or reduce_keys == "_all":
240             grouped_data = [("_all", input_data)]
241         else:
242             # Group the input data by the reduce key(s) while maintaining
243             original order
244             def get_group_key(item):
245                 key_values = []
246                 for key in reduce_keys:
247                     value = item[key]
248                     # Special handling for list-type values
249                     if isinstance(value, list):
250                         key_values.append(
251                             tuple(sorted(value)))
252                     ) # Convert list to sorted tuple
253                 else:
254                     key_values.append(value)
255             return tuple(key_values)
256
257             grouped_data = {}
258             for item in input_data:
259                 key = get_group_key(item)
260                 if key not in grouped_data:
261                     grouped_data[key] = []
262                     grouped_data[key].append(item)
263
264             # Convert the grouped data to a list of tuples
265             grouped_data = list(grouped_data.items())
266
267             def process_group(
268                 key: tuple, group_elems: list[dict]
269             ) -> tuple[dict | None, float]:
270                 if input_schema:
271                     group_list = [
272                         {k: item[k] for k in input_schema.keys() if k in
273                         item}

```

```

274         for item in group_elems
275     ]
276 else:
277     group_list = group_elems
278
279     total_cost = 0.0
280
281     # Apply value sampling if enabled
282     value_sampling = self.config.get("value_sampling", {})
283     if value_sampling.get("enabled", False):
284         sample_size = min(value_sampling["sample_size"],
285             len(group_list))
286         method = value_sampling["method"]
287
288         if method == "random":
289             group_sample = random.sample(group_list, sample_size)
290             group_sample.sort(key=lambda x: group_list.index(x))
291         elif method == "first_n":
292             group_sample = group_list[:sample_size]
293         elif method == "cluster":
294             group_sample, embedding_cost =
295             self._cluster_based_sampling(
296                 group_list, value_sampling, sample_size
297             )
298             group_sample.sort(key=lambda x: group_list.index(x))
299             total_cost += embedding_cost
300         elif method == "sem_sim":
301             group_sample, embedding_cost =
302             self._semantic_similarity_sampling(
303                 key, group_list, value_sampling, sample_size
304             )
305             group_sample.sort(key=lambda x: group_list.index(x))
306             total_cost += embedding_cost
307
308         group_list = group_sample
309
310         # Only execute merge-based plans if associative = True
311         if "merge_prompt" in self.config and
312             self.config.get("associative", True):
313             result, prompts, cost =
314             self._parallel_fold_and_merge(key, group_list)
315         elif self.config.get("fold_batch_size", None) and
316             self.config.get(
317                 "fold_batch_size"
318             ) >= len(group_list):
319                 # If the fold batch size is greater than or equal to the
320                 number of items in the group,
321                 # we can just run a single fold operation
322                 result, prompt, cost = self._batch_reduce(key,
323                     group_list)
324                 prompts = [prompt]
325             elif "fold_prompt" in self.config:
326                 result, prompts, cost = self._incremental_reduce(key,
327                     group_list)
328             else:
329                 result, prompt, cost = self._batch_reduce(key,
330                     group_list)
331                 prompts = [prompt]
332
333             total_cost += cost
334

```

```

335         # Add the counts of items in the group to the result
336         result[f"_counts_prereduce_{self.config['name']}"] =
337     len(group_elems)
338
339         if self.config.get("enable_observability", False):
340             # Add the _observability_{self.config['name']} key to the
341             result
342             result[f"_observability_{self.config['name']}"] =
343 {"prompts": prompts}
344
345             # Apply pass-through at the group level
346             if (
347                 result is not None
348                 and self.config.get("pass_through", False)
349                 and group_elems
350             ):
351                 for k, v in group_elems[0].items():
352                     if k not in self.config["output"]["schema"] and k not
353             in result:
354                         result[k] = v
355
356             # Add lineage information
357             if result is not None and self.lineage_keys:
358                 lineage = []
359                 for item in group_elems:
360                     lineage_item = {
361                         k: item.get(k) for k in self.lineage_keys if k in
362                         item
363                     }
364                     if lineage_item:
365                         lineage.append(lineage_item)
366                     result[f"{self.config['name']}__lineage"] = lineage
367
368             return result, total_cost
369
370         with ThreadPoolExecutor(max_workers=self.max_threads) as
371             executor:
372                 futures = [
373                     executor.submit(process_group, key, group)
374                     for key, group in grouped_data
375                 ]
376                 results = []
377                 total_cost = 0
378                 for future in rich_as_completed(
379                     futures,
380                     total=len(futures),
381                     desc=f"Processing {self.config['name']} (reduce) on all
382                     documents",
383                     leave=True,
384                     console=self.console,
385                 ):
386                     output, item_cost = future.result()
387                     total_cost += item_cost
388                     if output is not None:
389                         results.append(output)
390
391             if self.config.get("persist_intermediates", False):
392                 for result in results:
393                     key = tuple(result[k] for k in self.config["reduce_key"])
394                     if key in self.intermediates:
395                         result[f"__{self.config['name']}__intermediates"] = (

```

```

396             self.intermediates[key]
397         )
398
399     if self.status:
400         self.status.start()
401
402     return results, total_cost
403
404     def _cluster_based_sampling(
405         self, group_list: list[dict], value_sampling: dict, sample_size:
406         int
407     ) -> tuple[list[dict], float]:
408         if sample_size >= len(group_list):
409             return group_list, 0
410
411         clusters, cost = cluster_documents(
412             group_list, value_sampling, sample_size, self.runner.api
413         )
414
415         sampled_items = []
416         idx_added_already = set()
417         num_clusters = len(clusters)
418         for i in range(sample_size):
419             # Add a random item from the cluster
420             idx = i % num_clusters
421
422             # Skip if there are no items in the cluster
423             if len(clusters[idx]) == 0:
424                 continue
425
426             if len(clusters[idx]) == 1:
427                 # If there's only one item in the cluster, add it
428                 directly if we haven't already
429                 if idx not in idx_added_already:
430                     sampled_items.append(clusters[idx][0])
431                     continue
432
433             random_choice_idx = random.randint(0, len(clusters[idx]) - 1)
434             max_attempts = 10
435             while random_choice_idx in idx_added_already and max_attempts
436             > 0:
437                 random_choice_idx = random.randint(0, len(clusters[idx]))
438                 - 1)
439                 max_attempts -= 1
440                 idx_added_already.add(random_choice_idx)
441                 sampled_items.append(clusters[idx][random_choice_idx])
442
443         return sampled_items, cost
444
445     def _semantic_similarity_sampling(
446         self, key: tuple, group_list: list[dict], value_sampling: dict,
447         sample_size: int
448     ) -> tuple[list[dict], float]:
449         embedding_model = value_sampling["embedding_model"]
450         query_text = strict_render(
451             value_sampling["query_text"],
452             {"reduce_key": dict(zip(self.config["reduce_key"], key))},
453         )
454
455         embeddings, cost = get_embeddings_for_clustering(
456             group_list, value_sampling, self.runner.api

```

```

457     )
458
459     query_response = self.runner.api.gen_embedding(embedding_model,
460 [query_text])
461     query_embedding = query_response["data"][0]["embedding"]
462     cost += completion_cost(query_response)
463
464     from sklearn.metrics.pairwise import cosine_similarity
465
466     similarities = cosine_similarity([query_embedding], embeddings)
467 [0]
468
469     top_k_indices = np.argsort(similarities)[-sample_size:]
470
471     return [group_list[i] for i in top_k_indices], cost
472
473     def _parallel_fold_and_merge(
474         self, key: tuple, group_list: list[dict]
475     ) -> tuple[dict | None, float]:
476         """
477             Perform parallel folding and merging on a group of items.
478
479             This method implements a strategy that combines parallel folding
480             of input items
481             and merging of intermediate results to efficiently process large
482             groups. It works as follows:
483                 1. The input group is initially divided into smaller batches for
484                 efficient processing.
485                 2. The method performs an initial round of folding operations on
486                 these batches.
487                 3. After the first round of folds, a few merges are performed to
488                 estimate the merge runtime.
489                 4. Based on the estimated merge runtime and observed fold
490                 runtime, it calculates the optimal number of parallel folds. Subsequent
491                 rounds of folding are then performed concurrently, with the number of
492                 parallel folds determined by the runtime estimates.
493                 5. The folding process repeats in rounds, progressively reducing
494                 the number of items to be processed.
495                 6. Once all folding operations are complete, the method
496                 recursively performs final merges on the fold results to combine them
497                 into a final result.
498                 7. Throughout this process, the method may adjust the number of
499                 parallel folds based on updated performance metrics (i.e., fold and merge
500                 runtimes) to maintain efficiency.
501
502             Args:
503                 key (tuple): The reduce key tuple for the group.
504                 group_list (list[dict]): The list of items in the group to be
505             processed.
506
507             Returns:
508                 tuple[dict | None, float]: A tuple containing the final
509             merged result (or None if processing failed)
510                 and the total cost of the operation.
511         """
512         fold_batch_size = self.config["fold_batch_size"]
513         merge_batch_size = self.config["merge_batch_size"]
514         total_cost = 0
515         prompts = []
516
517         def calculate_num_parallel_folds():

```

```
518         fold_time, fold_default = self.get_fold_time()
519         merge_time, merge_default = self.get_merge_time()
520         num_group_items = len(group_list)
521         return (
522             max(
523                 1,
524                 int(
525                     (fold_time * num_group_items *
526                      math.log(merge_batch_size))
527                     / (fold_batch_size * merge_time)
528                     ),
529                 ),
530                 fold_default or merge_default,
531             )
532
533         num_parallel_folds, used_default_times =
534 calculate_num_parallel_folds()
535         fold_results = []
536         remaining_items = group_list
537
538         if self.config.get("persist_intermediates", False):
539             self.intermediates[key] = []
540             iter_count = 0
541
542             # Parallel folding and merging
543             with ThreadPoolExecutor(max_workers=self.max_threads) as
544 executor:
545                 while remaining_items:
546                     # Folding phase
547                     fold_futures = []
548                     for i in range(min(num_parallel_folds,
549 len(remaining_items))):
550                         batch = remaining_items[:fold_batch_size]
551                         remaining_items = remaining_items[fold_batch_size:]
552                         current_output = fold_results[i] if i <
553 len(fold_results) else None
554                         fold_futures.append(
555                             executor.submit(
556                                 self._increment_fold, key, batch,
557                                 current_output
558                             )
559                         )
560
561                     new_fold_results = []
562                     for future in as_completed(fold_futures):
563                         result, prompt, cost = future.result()
564                         total_cost += cost
565                         prompts.append(prompt)
566                         if result is not None:
567                             new_fold_results.append(result)
568                             if self.config.get("persist_intermediates",
569 False):
570                                 self.intermediates[key].append(
571                                     {
572                                         "iter": iter_count,
573                                         "intermediate": result,
574                                         "scratchpad": "
575                                         result["updated_scratchpad"],
576                                         }
577                                     )
578                                 iter_count += 1
```

```

579             # Update fold_results with new results
580             fold_results = new_fold_results +
581             fold_results[len(new_fold_results) :]
583
584             # Single pass merging phase
585             if (
586                 len(self.merge_times) < self.min_samples
587                 and len(fold_results) >= merge_batch_size
588             ):
589                 merge_futures = []
590                 for i in range(0, len(fold_results),
591                               merge_batch_size):
592                     batch = fold_results[i : i + merge_batch_size]
593                     merge_futures.append(
594                         executor.submit(self._merge_results, key,
595                                         batch)
596                     )
597
598                     new_results = []
599                     for future in as_completed(merge_futures):
600                         result, prompt, cost = future.result()
601                         total_cost += cost
602                         prompts.append(prompt)
603                         if result is not None:
604                             new_results.append(result)
605                             if self.config.get("persist_intermediates",
606                                False):
607                                 self.intermediates[key].append(
608                                     {
609                                         "iter": iter_count,
610                                         "intermediate": result,
611                                         "scratchpad": None,
612                                     }
613                                 )
614                                 iter_count += 1
615
616                     fold_results = new_results
617
618             # Recalculate num_parallel_folds if we used default times
619             if used_default_times:
620                 new_num_parallel_folds, used_default_times = (
621                     calculate_num_parallel_folds()
622                 )
623                 if not used_default_times:
624                     self.console.log(
625                         f"Recalculated num_parallel_folds from
626 {num_parallel_folds} to {new_num_parallel_folds}"
627                     )
628                     num_parallel_folds = new_num_parallel_folds
629
630             # Final merging if needed
631             while len(fold_results) > 1:
632                 self.console.log(f"Finished folding! Merging
633 {len(fold_results)} items.")
634                 with ThreadPoolExecutor(max_workers=self.max_threads) as
635                 executor:
636                     merge_futures = []
637                     for i in range(0, len(fold_results), merge_batch_size):
638                         batch = fold_results[i : i + merge_batch_size]
639                         merge_futures.append(

```

```

640             executor.submit(self._merge_results, key, batch)
641         )
642
643     new_results = []
644     for future in as_completed(merge_futures):
645         result, prompt, cost = future.result()
646         total_cost += cost
647         prompts.append(prompt)
648         if result is not None:
649             new_results.append(result)
650             if self.config.get("persist_intermediates",
651             False):
652                 self.intermediates[key].append(
653                     {
654                         "iter": iter_count,
655                         "intermediate": result,
656                         "scratchpad": None,
657                     }
658                 )
659                 iter_count += 1
660
661     fold_results = new_results
662
663     return (
664         (fold_results[0], prompts, total_cost)
665         if fold_results
666         else (None, prompts, total_cost)
667     )
668
669     def _incremental_reduce(
670         self, key: tuple, group_list: list[dict]
671     ) -> tuple[dict | None, list[str], float]:
672         """
673             Perform an incremental reduce operation on a group of items.
674
675             This method processes the group in batches, incrementally folding
676             the results.
677
678             Args:
679                 key (tuple): The reduce key tuple for the group.
680                 group_list (list[dict]): The list of items in the group to be
681             processed.
682
683             Returns:
684                 tuple[dict | None, list[str], float]: A tuple containing the
685             final reduced result (or None if processing failed),
686                 the list of prompts used, and the total cost of the
687             operation.
688             """
689             fold_batch_size = self.config["fold_batch_size"]
690             total_cost = 0
691             current_output = None
692             prompts = []
693
694             # Calculate and log the number of folds to be performed
695             num_folds = (len(group_list) + fold_batch_size - 1) //
696             fold_batch_size
697
698             scratchpad = ""
699             if self.config.get("persist_intermediates", False):
700                 self.intermediates[key] = []

```

```
701         iter_count = 0
702
703     for i in range(0, len(group_list), fold_batch_size):
704         # Log the current iteration and total number of folds
705         current_fold = i // fold_batch_size + 1
706         if self.config.get("verbose", False):
707             self.console.log(
708                 f"Processing fold {current_fold} of {num_folds} for
709 group with key {key}"
710                 )
711         batch = group_list[i : i + fold_batch_size]
712
713         folded_output, prompt, fold_cost = self._increment_fold(
714             key, batch, current_output, scratchpad
715         )
716         total_cost += fold_cost
717         prompts.append(prompt)
718
719     if folded_output is None:
720         continue
721
722     if self.config.get("persist_intermediates", False):
723         self.intermediates[key].append(
724             {
725                 "iter": iter_count,
726                 "intermediate": folded_output,
727                 "scratchpad":(
728                     folded_output.get("updated_scratchpad", ""),
729                     )
730                 )
731         iter_count += 1
732
733     # Pop off updated_scratchpad
734     if "updated_scratchpad" in folded_output:
735         scratchpad = folded_output["updated_scratchpad"]
736         if self.config.get("verbose", False):
737             self.console.log(
738                 f"Updated scratchpad for fold {current_fold}:
739 {scratchpad}"
740                 )
741         del folded_output["updated_scratchpad"]
742
743         current_output = folded_output
744
745     return current_output, prompts, total_cost
746
747     def validation_fn(self, response: dict[str, Any]):
748         structured_mode = (
749             self.config.get("output", {}).get("mode")
750             == OutputMode.STRUCTURED_OUTPUT.value
751         )
752         output = self.runner.api.parse_llm_response(
753             response,
754             schema=self.config["output"]["schema"],
755             use_structured_output=structured_mode,
756         )[0]
757         if self.runner.api.validate_output(self.config, output,
758             self.console):
759             return output, True
760         return output, False
761
```

```

762     def _increment_fold(
763         self,
764         key: tuple,
765         batch: list[dict],
766         current_output: dict | None,
767         scratchpad: str | None = None,
768     ) -> tuple[dict | None, str, float]:
769         """
770             Perform an incremental fold operation on a batch of items.
771
772             This method folds a batch of items into the current output using
773             the fold prompt.
774
775             Args:
776                 key (tuple): The reduce key tuple for the group.
777                 batch (list[dict]): The batch of items to be folded.
778                 current_output (dict | None): The current accumulated output,
779                 if any.
780                 scratchpad (str | None): The scratchpad to use for the fold
781                 operation.
782             Returns:
783                 tuple[dict | None, str, float]: A tuple containing the folded
784                 output (or None if processing failed),
785                 the prompt used, and the cost of the fold operation.
786             """
787             if current_output is None:
788                 return self._batch_reduce(key, batch, scratchpad)
789
790             start_time = time.time()
791             fold_prompt = strict_render(
792                 self.config["fold_prompt"],
793                 {
794                     "inputs": batch,
795                     "output": current_output,
796                     "reduce_key": dict(zip(self.config["reduce_key"], key)),
797                 },
798             )
799
800             response = self.runner.api.call_llm(
801                 self.config.get("model", self.default_model),
802                 "reduce",
803                 [{"role": "user", "content": fold_prompt}],
804                 self.config["output"]["schema"],
805                 scratchpad=scratchpad,
806                 timeout_seconds=self.config.get("timeout", 120),
807
808                 max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
809                 validation_config=(
810                     {
811                         "num_retries": self.num_retries_on_validate_failure,
812                         "val_rule": self.config.get("validate", []),
813                         "validation_fn": self.validation_fn,
814                     }
815                     if self.config.get("validate", None)
816                     else None
817                 ),
818                 bypass_cache=self.config.get("bypass_cache",
819                 self.bypass_cache),
820                 verbose=self.config.get("verbose", False),
821
822                 litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",

```

```

823     {}),
824         op_config=self.config,
825     )
826
827     end_time = time.time()
828     self._update_fold_time(end_time - start_time)
829
830     if response.validated:
831         structured_mode = (
832             self.config.get("output", {}).get("mode")
833             == OutputMode.STRUCTURED_OUTPUT.value
834         )
835         folded_output = self.runner.api.parse_llm_response(
836             response.response,
837             schema=self.config["output"]["schema"],
838             manually_fix_errors=self.manually_fix_errors,
839             use_structured_output=structured_mode,
840         )[0]
841
842         folded_output.update(dict(zip(self.config["reduce_key"], key)))
843         fold_cost = response.total_cost
844
845         return folded_output, fold_prompt, fold_cost
846
847     return None, fold_prompt, fold_cost
848
849
850     def _merge_results(
851         self, key: tuple, outputs: list[dict]
852     ) -> tuple[dict | None, str, float]:
853         """
854             Merge multiple outputs into a single result.
855
856             This method merges a list of outputs using the merge prompt.
857
858             Args:
859                 key (tuple): The reduce key tuple for the group.
860                 outputs (list[dict]): The list of outputs to be merged.
861
862             Returns:
863                 tuple[dict | None, str, float]: A tuple containing the merged
864                 output (or None if processing failed),
865                 the prompt used, and the cost of the merge operation.
866         """
867         start_time = time.time()
868         merge_prompt = strict_render(
869             self.config["merge_prompt"],
870             {
871                 "outputs": outputs,
872                 "reduce_key": dict(zip(self.config["reduce_key"], key)),
873             },
874         )
875         response = self.runner.api.call_llm(
876             self.config.get("model", self.default_model),
877             "merge",
878             [{"role": "user", "content": merge_prompt}],
879             self.config["output"]["schema"],
880             timeout_seconds=self.config.get("timeout", 120),
881
882             max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
883             validation_config=(

```

```

884         {
885             "num_retries": self.num_retries_on_validate_failure,
886             "val_rule": self.config.get("validate", []),
887             "validation_fn": self.validation_fn,
888         }
889         if self.config.get("validate", None)
890             else None
891         ),
892         bypass_cache=self.config.get("bypass_cache",
893         self.bypass_cache),
894         verbose=self.config.get("verbose", False),
895
896         litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
897         {}),
898         op_config=self.config,
899     )
900
901     end_time = time.time()
902     self._update_merge_time(end_time - start_time)
903
904     if response.validated:
905         structured_mode = (
906             self.config.get("output", {}).get("mode")
907             == OutputMode.STRUCTURED_OUTPUT.value
908         )
909         merged_output = self.runner.api.parse_llm_response(
910             response.response,
911             schema=self.config["output"]["schema"],
912             manually_fix_errors=self.manually_fix_errors,
913             use_structured_output=structured_mode,
914         )[0]
915         merged_output.update(dict(zip(self.config["reduce_key"],
916         key)))
917         merge_cost = response.total_cost
918         return merged_output, merge_prompt, merge_cost
919
920     return None, merge_prompt, merge_cost
921
922     def get_fold_time(self) -> tuple[float, bool]:
923         """
924             Get the average fold time or a default value.
925
926             Returns:
927                 tuple[float, bool]: A tuple containing the average fold time
928             (or default) and a boolean
929                 indicating whether the default value was used.
930
931             if "fold_time" in self.config:
932                 return self.config["fold_time"], False
933             with self.lock:
934                 if len(self.fold_times) >= self.min_samples:
935                     return sum(self.fold_times) / len(self.fold_times), False
936             return 1.0, True # Default to 1 second if no data is available
937
938             def get_merge_time(self) -> tuple[float, bool]:
939
940                 Get the average merge time or a default value.
941
942                 Returns:
943                     tuple[float, bool]: A tuple containing the average merge time
944             (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__(*, **kwargs)`

Initialize the ReduceOperation.

Parameters:

Name	Type	Description	Default
<code>*args</code>		Variable length argument list.	<code>()</code>
<code>**kwargs</code>		Arbitrary keyword arguments.	<code>{}</code>

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

```

159     def __init__(self, *args, **kwargs):
160         """
161             Initialize the ReduceOperation.
162
163             Args:
164                 *args: Variable length argument list.
165                 **kwargs: Arbitrary keyword arguments.
166
167             super().__init__(*args, **kwargs)
168             self.min_samples = 5
169             self.max_samples = 1000
170             self.fold_times = deque(maxlen=self.max_samples)
171             self.merge_times = deque(maxlen=self.max_samples)
172             self.lock = Lock()
173             self.config["reduce_key"] = (
174                 [self.config["reduce_key"]]
175                 if isinstance(self.config["reduce_key"], str)
176                 else self.config["reduce_key"]
177             )
178             self.intermediates = {}
179             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
<code>input_data</code>	<code>list[dict]</code>	The input data to process.	<i>required</i>

Returns:

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

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

```
181     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
182         """
183             Execute the reduce operation on the provided input data.
184
185             This method sorts and groups the input data by the reduce key(s),
186             then processes each group
187             using either parallel fold and merge, incremental reduce, or batch
188             reduce strategies.
189
190             Args:
191                 input_data (list[dict]): The input data to process.
192
193             Returns:
194                 tuple[list[dict], float]: A tuple containing the processed
195                 results and the total cost of the operation.
196             """
197             if self.config.get("gleaning", {}).get("validation_prompt", None):
198                 self.console.log(
199                     f"Using gleaning with validation prompt:
200 {self.config.get('gleaning', {}).get('validation_prompt', '')}"
201                 )
202
203             reduce_keys = self.config["reduce_key"]
204             if isinstance(reduce_keys, str):
205                 reduce_keys = [reduce_keys]
206             input_schema = self.config.get("input", {}).get("schema", {})
207
208             if self.status:
209                 self.status.stop()
210
211             # Check if we need to group everything into one group
212             if reduce_keys == ["_all"] or reduce_keys == "_all":
213                 grouped_data = [("_all", input_data)]
214             else:
215                 # Group the input data by the reduce key(s) while maintaining
216                 original order
217                 def get_group_key(item):
218                     key_values = []
219                     for key in reduce_keys:
220                         value = item[key]
221                         # Special handling for list-type values
222                         if isinstance(value, list):
223                             key_values.append(
224                                 tuple(sorted(value))
225                             ) # Convert list to sorted tuple
226                         else:
227                             key_values.append(value)
228                     return tuple(key_values)
229
230             grouped_data = {}
231             for item in input_data:
232                 key = get_group_key(item)
233                 if key not in grouped_data:
234                     grouped_data[key] = []
235                     grouped_data[key].append(item)
236
237             # Convert the grouped data to a list of tuples
```

```

238     grouped_data = list(grouped_data.items())
239
240     def process_group(
241         key: tuple, group_elems: list[dict]
242     ) -> tuple[dict | None, float]:
243         if input_schema:
244             group_list = [
245                 {k: item[k] for k in input_schema.keys() if k in item}
246                 for item in group_elems
247             ]
248         else:
249             group_list = group_elems
250
251         total_cost = 0.0
252
253         # Apply value sampling if enabled
254         value_sampling = self.config.get("value_sampling", {})
255         if value_sampling.get("enabled", False):
256             sample_size = min(value_sampling["sample_size"],
257             len(group_list))
258             method = value_sampling["method"]
259
260             if method == "random":
261                 group_sample = random.sample(group_list, sample_size)
262                 group_sample.sort(key=lambda x: group_list.index(x))
263             elif method == "first_n":
264                 group_sample = group_list[:sample_size]
265             elif method == "cluster":
266                 group_sample, embedding_cost =
267             self._cluster_based_sampling(
268                 group_list, value_sampling, sample_size
269             )
270             group_sample.sort(key=lambda x: group_list.index(x))
271             total_cost += embedding_cost
272             elif method == "sem_sim":
273                 group_sample, embedding_cost =
274             self._semantic_similarity_sampling(
275                 key, group_list, value_sampling, sample_size
276             )
277             group_sample.sort(key=lambda x: group_list.index(x))
278             total_cost += embedding_cost
279
280             group_list = group_sample
281
282             # Only execute merge-based plans if associative = True
283             if "merge_prompt" in self.config and
284             self.config.get("associative", True):
285                 result, prompts, cost = self._parallel_fold_and_merge(key,
286             group_list)
287                 elif self.config.get("fold_batch_size", None) and
288             self.config.get(
289                 "fold_batch_size"
290             ) >= len(group_list):
291                 # If the fold batch size is greater than or equal to the
292                 number of items in the group,
293                     # we can just run a single fold operation
294                     result, prompt, cost = self._batch_reduce(key, group_list)
295                     prompts = [prompt]
296                     elif "fold_prompt" in self.config:
297                         result, prompts, cost = self._incremental_reduce(key,
298             group_list)

```

```

299     else:
300         result, prompt, cost = self._batch_reduce(key, group_list)
301         prompts = [prompt]
302
303         total_cost += cost
304
305         # Add the counts of items in the group to the result
306         result[f"_counts_prereduce_{self.config['name']}"] =
307         len(group_elems)
308
309         if self.config.get("enable_observability", False):
310             # Add the _observability_{self.config['name']} key to the
311             result
312             result[f"_observability_{self.config['name']}"] = {"prompts":'
313             prompts}
314
315         # Apply pass-through at the group level
316         if (
317             result is not None
318             and self.config.get("pass_through", False)
319             and group_elems
320         ):
321             for k, v in group_elems[0].items():
322                 if k not in self.config["output"]["schema"] and k not in
323             result:
324                 result[k] = v
325
326         # Add lineage information
327         if result is not None and self.lineage_keys:
328             lineage = []
329             for item in group_elems:
330                 lineage_item = {
331                     k: item.get(k) for k in self.lineage_keys if k in
332                 item
333                 }
334                 if lineage_item:
335                     lineage.append(lineage_item)
336             result[f"{self.config['name']}_lineage"] = lineage
337
338         return result, total_cost
339
340     with ThreadPoolExecutor(max_workers=self.max_threads) as executor:
341         futures = [
342             executor.submit(process_group, key, group)
343             for key, group in grouped_data
344         ]
345         results = []
346         total_cost = 0
347         for future in rich_as_completed(
348             futures,
349             total=len(futures),
350             desc=f"Processing {self.config['name']} (reduce) on all
351             documents",
352             leave=True,
353             console=self.console,
354         ):
355             output, item_cost = future.result()
356             total_cost += item_cost
357             if output is not None:
358                 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
824             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
839         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`

```
522     class ParallelMapOperation(BaseOperation):
523         class schema(BaseOperation.schema):
524             type: str = "parallel_map"
525             prompts: list[dict[str, Any]] | None = None
526             output: dict[str, Any] | None = None
527             drop_keys: list[str] | None = None
528             enable_observability: bool = False
529             pdf_url_key: str | None = None
530
531             @field_validator("prompts")
532             def validate_prompts(cls, v):
533                 if v is not None:
534                     if not v:
535                         raise ValueError("The 'prompts' list cannot be
empty")
536
537                     for i, prompt_config in enumerate(v):
538                         # Validate required keys exist
539                         if "prompt" not in prompt_config:
540                             raise ValueError(
541                                 f"Missing required key 'prompt' in prompt
configuration {i}")
542
543                         if "output_keys" not in prompt_config:
544                             raise ValueError(
545                                 f"Missing required key 'output_keys' in
prompt configuration {i}")
546
547                         # Validate output_keys is not empty
548                         if not prompt_config["output_keys"]:
549                             raise ValueError(
550                                 f"'output_keys' list in prompt configuration
{i} cannot be empty")
551
552                         # Check if the prompt is a valid Jinja2 template
553                         try:
554                             Template(prompt_config["prompt"])
555                         except Exception as e:
556                             raise ValueError(
557                                 f"Invalid Jinja2 template in prompt
configuration {i}: {str(e)}")
558                         ) from e
559
560                     return v
561
562             @model_validator(mode="after")
563             def validate_prompt_requirements(self):
564                 # If drop_keys is not specified, prompts must be present
565                 if not self.drop_keys and not self.prompts:
566                     raise ValueError(
567                         "If 'drop_keys' is not specified, 'prompts' must be
present in the configuration")
568
569                 # Check if all output schema keys are covered by the prompts
570                 if self.prompts and self.output and "schema" in self.output:
```

```

579         output_schema = self.output["schema"]
580         output_keys_covered = set()
581         for prompt_config in self.prompts:
582
583             output_keys_covered.update(prompt_config["output_keys"])
584
585             missing_keys = set(output_schema.keys()) -
586             output_keys_covered
587             if missing_keys:
588                 raise ValueError(
589                     f"The following output schema keys are not
590 covered by any prompt: {missing_keys}"
591                 )
592
593         return self
594
595     def __init__(
596         self,
597         *args,
598         **kwargs,
599     ):
600         super().__init__(*args, **kwargs)
601
602     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
603         """
604             Executes the parallel map operation on the provided input data.
605
606             Args:
607                 input_data (list[dict]): The input data to process.
608
609             Returns:
610                 tuple[list[dict], float]: A tuple containing the processed
611             results and the total cost of the operation.
612
613             This method performs the following steps:
614             1. If prompts are specified, it processes each input item using
615             multiple prompts in parallel
616             2. Aggregates results from different prompts for each input item
617             3. Validates the combined output for each item
618             4. If drop_keys is specified, it drops the specified keys from
619             each document
620             5. Calculates total cost of the operation
621         """
622
623         results = []
624         total_cost = 0
625         output_schema = self.config.get("output", {}).get("schema", {})
626
627         # Check if there's no prompt and only drop_keys
628         if "prompts" not in self.config and "drop_keys" in self.config:
629             # If only drop_keys is specified, simply drop the keys and
630             return
631             dropped_results = []
632             for item in input_data:
633                 new_item = {
634                     k: v for k, v in item.items() if k not in
635                     self.config["drop_keys"]
636                 }
637                 dropped_results.append(new_item)
638             return dropped_results, 0.0 # Return the modified data with
639             no cost

```

```
640         if self.status:
641             self.status.stop()
642
643
644     def process_prompt(item, prompt_config):
645         prompt = strict_render(prompt_config["prompt"], {"input":
646             item})
647
648         messages = [{"role": "user", "content": prompt}]
649         if self.config.get("pdf_url_key", None):
650             try:
651                 pdf_url = item[self.config["pdf_url_key"]]
652             except KeyError:
653                 raise ValueError(
654                     f"PDF URL key '{self.config['pdf_url_key']}' not
655                     found in input data")
656
657             # Download content
658             if pdf_url.startswith("http"):
659                 file_data = requests.get(pdf_url).content
660             else:
661                 with open(pdf_url, "rb") as f:
662                     file_data = f.read()
663             encoded_file = base64.b64encode(file_data).decode("utf-
664             8")
665             base64_url = f"data:application/pdf;base64,
666             {encoded_file}"
667
668             messages[0]["content"] = [
669                 {"type": "image_url", "image_url": {"url":
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_kw_args=self.config.get(
696                     "litellm_completion_kw_args", {})
697                 ),
698                 op_config=self.config,
699             )
700             structured_mode = (
```

```

701         self.config.get("output", {}).get("mode")
702         == OutputMode.STRUCTURED_OUTPUT.value
703     )
704     output = self.runner.api.parse_llm_response(
705         response.response,
706         schema=local_output_schema,
707         tools=prompt_config.get("tools", None),
708         manually_fix_errors=self.manually_fix_errors,
709         use_structured_output=structured_mode,
710     )[0]
711     return output, prompt, response.total_cost
712
713     with ThreadPoolExecutor(max_workers=self.max_threads) as
714     executor:
715         if "prompts" in self.config:
716             # Create all futures at once
717             all_futures = [
718                 executor.submit(process_prompt, item, prompt_config)
719                 for item in input_data
720                 for prompt_config in self.config["prompts"]
721             ]
722
723             # Process results in order
724             for i in tqdm(
725                 range(len(all_futures)),
726                 desc="Processing parallel map items",
727             ):
728                 future = all_futures[i]
729                 output, prompt, cost = future.result()
730                 total_cost += cost
731
732                 # Determine which item this future corresponds to
733                 item_index = i // len(self.config["prompts"])
734                 prompt_index = i % len(self.config["prompts"])
735
736                 # Initialize or update the item_result
737                 if prompt_index == 0:
738                     item_result = input_data[item_index].copy()
739                     results[item_index] = item_result
740
741                 # Fetch the item_result
742                 item_result = results[item_index]
743
744                 if self.config.get("enable_observability", False):
745                     if f"_observability_{self.config['name']} not in
item_result:
746
747                     item_result[f"_observability_{self.config['name']}"] = {}
748
749                     item_result[f"_observability_{self.config['name']}"].update(
750                         {f"prompt_{prompt_index}": prompt}
751                     )
752
753                     # Update the item_result with the output
754                     item_result.update(output)
755
756             else:
757                 results = {i: item.copy() for i, item in
enumerate(input_data)}
758
759                 # 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.	<i>required</i>

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`

```
593     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
594         """
595             Executes the parallel map operation on the provided input data.
596
597             Args:
598                 input_data (list[dict]): The input data to process.
599
600             Returns:
601                 tuple[list[dict], float]: A tuple containing the processed
602                 results and the total cost of the operation.
603
604             This method performs the following steps:
605                 1. If prompts are specified, it processes each input item using
606                     multiple prompts in parallel
607                 2. Aggregates results from different prompts for each input item
608                 3. Validates the combined output for each item
609                 4. If drop_keys is specified, it drops the specified keys from each
610                     document
611                 5. Calculates total cost of the operation
612         """
613         results = []
614         total_cost = 0
615         output_schema = self.config.get("output", {}).get("schema", {})
616
617         # Check if there's no prompt and only drop_keys
618         if "prompts" not in self.config and "drop_keys" in self.config:
619             # If only drop_keys is specified, simply drop the keys and return
620             dropped_results = []
621             for item in input_data:
622                 new_item = {
623                     k: v for k, v in item.items() if k not in
624                     self.config["drop_keys"]
625                 }
626                 dropped_results.append(new_item)
627             return dropped_results, 0.0 # Return the modified data with no
628             cost
629
630             if self.status:
631                 self.status.stop()
632
633             def process_prompt(item, prompt_config):
634                 prompt = strict_render(prompt_config["prompt"], {"input": item})
635                 messages = [{"role": "user", "content": prompt}]
636                 if self.config.get("pdf_url_key", None):
637                     try:
638                         pdf_url = item[self.config["pdf_url_key"]]
639                     except KeyError:
640                         raise ValueError(
641                             f"PDF URL key '{self.config['pdf_url_key']}' not
642                             found in input data"
643                         )
644                         # Download content
645                         if pdf_url.startswith("http"):
646                             file_data = requests.get(pdf_url).content
647                         else:
648                             with open(pdf_url, "rb") as f:
649                                 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",

```

```

711     ):
712         future = all_futures[i]
713         output, prompt, cost = future.result()
714         total_cost += cost
715
716         # Determine which item this future corresponds to
717         item_index = i // len(self.config["prompts"])
718         prompt_index = i % len(self.config["prompts"])
719
720         # Initialize or update the item_result
721         if prompt_index == 0:
722             item_result = input_data[item_index].copy()
723             results[item_index] = item_result
724
725         # Fetch the item_result
726         item_result = results[item_index]
727
728         if self.config.get("enable_observability", False):
729             if f"_observability_{self.config['name']} not in
730 item_result:
731
732             item_result[f"_observability_{self.config['name']}"] = {}
733
734             item_result[f"_observability_{self.config['name']}"].update(
735                 {f"prompt_{prompt_index}": prompt}
736             )
737
738             # Update the item_result with the output
739             item_result.update(output)
740
741         else:
742             results = {i: item.copy() for i, item in
743 enumerate(input_data)}
744
745         # Apply drop_keys if specified
746         if "drop_keys" in self.config:
747             drop_keys = self.config["drop_keys"]
748             for item in results.values():
749                 for key in drop_keys:
750                     item.pop(key, None)
751
752         if self.status:
753             self.status.start()
754
755         # Return the results in order
756         return [results[i] for i in range(len(input_data)) if i in results],
757         total_cost

```

`docetl.operations.filter.FilterOperation`

Bases: [MapOperation](#)

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

```
10  class FilterOperation(MapOperation):
11      class schema(MapOperation.schema):
12          type: str = "filter"
13          prompt: str
14          output: dict[str, Any]
15
16      @model_validator(mode="after")
17      def validate_filter_output_schema(self):
18          # Check that schema exists and has the right structure for
19          filtering
20          schema_dict = self.output["schema"]
21
22          # Filter out _short_explanation for validation
23          schema = {k: v for k, v in schema_dict.items() if k !=
24          "_short_explanation"}
25          if len(schema) != 1:
26              raise ValueError(
27                  "The 'schema' in 'output' configuration must have
28                  exactly one key-value pair that maps to a boolean value"
29              )
30
31          key, value = next(iter(schema.items()))
32          if value not in ["bool", "boolean"]:
33              raise TypeError(
34                  f"The value in the 'schema' must be of type bool, got
35 {value}"
36              )
37
38          return self
39
40      def execute(
41          self, input_data: list[dict], is_build: bool = False
42      ) -> tuple[list[dict], float]:
43          """
44              Executes the filter operation on the input data.
45
46              Args:
47                  input_data (list[dict]): A list of dictionaries to process.
48                  is_build (bool): Whether the operation is being executed in
49              the build phase. Defaults to False.
50
51              Returns:
52                  tuple[list[dict], float]: A tuple containing the filtered
53              list of dictionaries
54                  and the total cost of the operation.
55
56              This method performs the following steps:
57              1. Processes each input item using an LLM model
58              2. Validates the output
59              3. Filters the results based on the specified filter key
60              4. Calculates the total cost of the operation
61
62              The method uses multi-threading to process items in parallel,
63              improving performance
64              for large datasets.
65
66              Usage:
```

```

67     ````python
68     from docetl.operations import FilterOperation
69
70     config = {
71         "prompt": "Determine if the following item is important:
72 {{input}}",
73         "output": {
74             "schema": {"is_important": "bool"}
75         },
76         "model": "gpt-3.5-turbo"
77     }
78     filter_op = FilterOperation(config)
79     input_data = [
80         {"id": 1, "text": "Critical update"},
81         {"id": 2, "text": "Regular maintenance"}
82     ]
83     results, cost = filter_op.execute(input_data)
84     print(f"Filtered results: {results}")
85     print(f"Total cost: {cost}")
86     ```
87     """
88     filter_key = next(
89         iter(
90             [
91                 k
92                     for k in self.config["output"]["schema"].keys()
93                     if k != "_short_explanation"
94             ]
95         )
96     )
97
98     results, total_cost = super().execute(input_data)
99
100    # Drop records with filter_key values that are False
101    if not is_build:
102        results = [result for result in results if
103 result[filter_key]]
104
105        # Drop the filter_key from the results
106        for result in results:
107            result.pop(filter_key, None)
108
109    return results, total_cost

```

`execute(input_data, is_build=False)`

Executes the filter operation on the input data.

Parameters:

Name	Type	Description	Default
<code>input_data</code>	<code>list[dict]</code>	A list of dictionaries to process.	<i>required</i>

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`

```
36     def execute(
37         self, input_data: list[dict], is_build: bool = False
38     ) -> tuple[list[dict], float]:
39         """
40             Executes the filter operation on the input data.
41
42             Args:
43                 input_data (list[dict]): A list of dictionaries to process.
44                 is_build (bool): Whether the operation is being executed in the
45             build phase. Defaults to False.
46
47             Returns:
48                 tuple[list[dict], float]: A tuple containing the filtered list of
49             dictionaries
50                 and the total cost of the operation.
51
52             This method performs the following steps:
53             1. Processes each input item using an LLM model
54             2. Validates the output
55             3. Filters the results based on the specified filter key
56             4. Calculates the total cost of the operation
57
58             The method uses multi-threading to process items in parallel,
59             improving performance
60             for large datasets.
61
62             Usage:
63             ```python
64             from docetl.operations import FilterOperation
65
66             config = {
67                 "prompt": "Determine if the following item is important:
68             {{input}}",
69                 "output": {
70                     "schema": {"is_important": "bool"}
71                 },
72                 "model": "gpt-3.5-turbo"
73             }
74             filter_op = FilterOperation(config)
75             input_data = [
76                 {"id": 1, "text": "Critical update"},
77                 {"id": 2, "text": "Regular maintenance"}
78             ]
79             results, cost = filter_op.execute(input_data)
80             print(f"Filtered results: {results}")
81             print(f"Total cost: {cost}")
82             ```
83
84             """
85             filter_key = next(
86                 iter(
87                     [
88                         k
89                         for k in self.config["output"]["schema"].keys()
90                         if k != "_short_explanation"
91                     ]
92                 )
93             )
```

```
93     results, total_cost = super().execute(input_data)
94
95     # Drop records with filter_key values that are False
96     if not is_build:
97         results = [result for result in results if result[filter_key]]
98
99     # Drop the filter_key from the results
100    for result in results:
101        result.pop(filter_key, None)
102
103    return results, total_cost
```

`docetl.operations.equijoin.EquijoinOperation`

Bases: `BaseOperation`

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

```
56  class EquijoinOperation(BaseOperation):
57      class schema(BaseOperation.schema):
58          type: str = "equijoin"
59          comparison_prompt: str
60          output: dict[str, Any] | None = None
61          blocking_threshold: float | None = None
62          blocking_conditions: list[str] | None = None
63          limits: dict[str, int] | None = None
64          comparison_model: str | None = None
65          optimize: bool | None = None
66          embedding_model: str | None = None
67          embedding_batch_size: int | None = None
68          compare_batch_size: int | None = None
69          limit_comparisons: int | None = None
70          blocking_keys: dict[str, list[str]] | None = None
71          timeout: int | None = None
72          litellm_completion_kwargs: dict[str, Any] = {}
73
74  @field_validator("blocking_keys")
75  def validate_blocking_keys(cls, v):
76      if v is not None:
77          if "left" not in v or "right" not in v:
78              raise ValueError(
79                  "Both 'left' and 'right' must be specified in
80 'blocking_keys'"
81              )
82      return v
83
84  @field_validator("limits")
85  def validate_limits(cls, v):
86      if v is not None:
87          if "left" not in v or "right" not in v:
88              raise ValueError(
89                  "Both 'left' and 'right' must be specified in
90 'limits'"
91              )
92      return v
93
94  def compare_pair(
95      self,
96      comparison_prompt: str,
97      model: str,
98      item1: dict,
99      item2: dict,
100     timeout_seconds: int = 120,
101     max_retries_per_timeout: int = 2,
102 ) -> tuple[bool, float]:
103     """
104         Compares two items using an LLM model to determine if they match.
105
106     Args:
107         comparison_prompt (str): The prompt template for comparison.
108         model (str): The LLM model to use for comparison.
109         item1 (dict): The first item to compare.
110         item2 (dict): The second item to compare.
111         timeout_seconds (int): The timeout for the LLM call in
112         seconds.
```

```

113             max_retries_per_timeout (int): The maximum number of retries
114             per timeout.
115
116             Returns:
117                 tuple[bool, float]: A tuple containing a boolean indicating
118                 whether the items match and the cost of the comparison.
119                 """
120
121             try:
122                 prompt = strict_render(comparison_prompt, {"left": item1,
123 "right": item2})
124                 except Exception as e:
125                     self.console.log(f"[red]Error rendering prompt: {e}[/red]")
126                     return False, 0
127                 response = self.runner.api.call_llm(
128                     model,
129                     "compare",
130                     [{"role": "user", "content": prompt}],
131                     {"is_match": "bool"},
132                     timeout_seconds=timeout_seconds,
133                     max_retries_per_timeout=max_retries_per_timeout,
134                     bypass_cache=self.config.get("bypass_cache",
135                     self.bypass_cache),
136
137                     litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
138                     {}),
139                     op_config=self.config,
140                     )
141                     cost = 0
142                     try:
143                         cost = response.total_cost
144                         output = self.runner.api.parse_llm_response(
145                             response.response, {"is_match": "bool"}
146                             )[0]
147                         except Exception as e:
148                             self.console.log(f"[red]Error parsing LLM response: {e}[/red]")
149
150                         return False, cost
151                     return output["is_match"], cost
152
153             def execute(
154                 self, left_data: list[dict], right_data: list[dict]
155                 ) -> tuple[list[dict], float]:
156                 """
157                     Executes the equijoin operation on the provided datasets.
158
159                     Args:
160                         left_data (list[dict]): The left dataset to join.
161                         right_data (list[dict]): The right dataset to join.
162
163                     Returns:
164                         tuple[list[dict], float]: A tuple containing the joined
165                         results and the total cost of the operation.
166
167                     Usage:
168                     ```python
169                     from docetl.operations import EquijoinOperation
170
171                     config = {
172                         "blocking_keys": {
173                             "left": ["id"],

```

```

174         "right": ["user_id"]
175     },
176     "limits": {
177         "left": 1,
178         "right": 1
179     },
180     "comparison_prompt": "Compare {{left}} and {{right}} and
determine if they match.",
181     "blocking_threshold": 0.8,
182     "blocking_conditions": ["left['id'] == right['user_id']"],
183     "limit_comparisons": 1000
184   }
185   equijoin_op = EquijoinOperation(config)
186   left_data = [{"id": 1, "name": "Alice"}, {"id": 2, "name":
187 "Bob"}]
188   right_data = [{"user_id": 1, "age": 30}, {"user_id": 2, "age": 189
190 25}]
191   results, cost = equijoin_op.execute(left_data, right_data)
192   print(f"Joined results: {results}")
193   print(f"Total cost: {cost}")
194   ```

195
196   This method performs the following steps:
197   1. Initial blocking based on specified conditions (if any)
198   2. Embedding-based blocking (if threshold is provided)
199   3. LLM-based comparison for blocked pairs
200   4. Result aggregation and validation
201
202   The method also calculates and logs statistics such as
203   comparisons saved by blocking and join selectivity.
204   """
205
206   blocking_keys = self.config.get("blocking_keys", {})
207   left_keys = blocking_keys.get(
208       "left", list(left_data[0].keys()) if left_data else []
209   )
210   right_keys = blocking_keys.get(
211       "right", list(right_data[0].keys()) if right_data else []
212   )
213   limits = self.config.get(
214       "limits", {"left": float("inf"), "right": float("inf")})
215   )
216   left_limit = limits["left"]
217   right_limit = limits["right"]
218   blocking_threshold = self.config.get("blocking_threshold")
219   blocking_conditions = self.config.get("blocking_conditions", [])
220   limit_comparisons = self.config.get("limit_comparisons")
221   total_cost = 0
222
223   if len(left_data) == 0 or len(right_data) == 0:
224       return [], 0
225
226   if self.status:
227       self.status.stop()
228
229   # Initial blocking using multiprocessing
230   num_processes = min(cpu_count(), len(left_data))
231
232   self.console.log(
233       f"Starting to run code-based blocking rules for
{len(left_data)} left and {len(right_data)} right rows ({len(left_data)} *
234

```

```

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

236     )  

237  

238     with Pool(  

239         processes=num_processes,  

240         initializer=init_worker,  

241         initargs=(right_data, blocking_conditions),  

242     ) as pool:  

243         blocked_pairs_nested = pool.map(process_left_item, left_data)  

244  

245         # Flatten the nested list of blocked pairs  

246         blocked_pairs = [pair for sublist in blocked_pairs_nested for  

247             pair in sublist]  

248  

249         # Check if we have exceeded the pairwise comparison limit  

250         if limit_comparisons is not None and len(blocked_pairs) >  

251             limit_comparisons:  

252             # Sample pairs based on cardinality and length  

253             sampled_pairs = stratified_length_sample(  

254                 blocked_pairs, limit_comparisons, sample_size=1000,  

255                 console=self.console  

256             )  

257  

258             # Calculate number of dropped pairs  

259             dropped_pairs = len(blocked_pairs) - limit_comparisons  

260  

261             # Prompt the user for confirmation  

262             if self.status:  

263                 self.status.stop()  

264             if not Confirm.ask(  

265                 f"[yellow]Warning: {dropped_pairs} pairs will be dropped  

266                 due to the comparison limit.  

267                 Proceeding with {limit_comparisons} randomly sampled  

268                 pairs. "  

269                 f"Do you want to continue?[/yellow]",  

270                 console=self.console,  

271             ):  

272                 raise ValueError("Operation cancelled by user due to pair  

273                 limit.")  

274  

275             if self.status:  

276                 self.status.start()  

277  

278             blocked_pairs = sampled_pairs  

279  

280             self.console.log(  

281                 f"Number of blocked pairs after initial blocking:  

282                 {len(blocked_pairs)}")  

283             )  

284  

285             if blocking_threshold is not None:  

286                 embedding_model = self.config.get("embedding_model",  

287                     self.default_model)  

288                 model_input_context_length = model_cost.get(embedding_model,  

289                     {}).get(  

290                         "max_input_tokens", 8192  

291                     )  

292  

293             def get_embeddings(  

294                 input_data: list[dict[str, Any]], keys: list[str], name:  

295                 str

```

```

296         ) -> tuple[list[list[float]], float]:
297             texts = [
298                 " ".join(str(item[key])) for key in keys if key in
299             item)[
300                 : model_input_context_length * 4
301             ]
302             for item in input_data
303         ]
304
305             embeddings = []
306             total_cost = 0
307             batch_size = 2000
308             for i in range(0, len(texts), batch_size):
309                 batch = texts[i : i + batch_size]
310                 self.console.log(
311                     f"On iteration {i} for creating embeddings for
312             {name} data"
313                 )
314                 response = self.runner.api.gen_embedding(
315                     model=embedding_model,
316                     input=batch,
317                     )
318                 embeddings.extend([data["embedding"] for data in
319             response["data"]])
320                 total_cost += completion_cost(response)
321             return embeddings, total_cost
322
323             left_embeddings, left_cost = get_embeddings(left_data,
324             left_keys, "left")
325             right_embeddings, right_cost = get_embeddings(
326                 right_data, right_keys, "right"
327             )
328             total_cost += left_cost + right_cost
329             self.console.log(
330                 f"Created embeddings for datasets. Total embedding
331             creation cost: {total_cost}"
332             )
333
334             # Compute all cosine similarities in one call
335             from sklearn.metrics.pairwise import cosine_similarity
336
337             similarities = cosine_similarity(left_embeddings,
338             right_embeddings)
339
340             # Additional blocking based on embeddings
341             # Find indices where similarity is above threshold
342             above_threshold = np.argwhere(similarities >=
343             blocking_threshold)
344             self.console.log(
345                 f"There are {above_threshold.shape[0]} pairs above the
346             threshold."
347             )
348             block_pair_set = set(
349                 (get_hashable_key(left_item),
350             get_hashable_key(right_item))
351                 for left_item, right_item in blocked_pairs
352             )
353
354             # If limit_comparisons is set, take only the top pairs
355             if limit_comparisons is not None:
356                 # First, get all pairs above threshold

```

```

357         above_threshold_pairs = [(int(i), int(j)) for i, j in
358             above_threshold]
359
360             # Sort these pairs by their similarity scores
361             sorted_pairs = sorted(
362                 above_threshold_pairs,
363                 key=lambda pair: similarities[pair[0], pair[1]],
364                 reverse=True,
365             )
366
367             # Take the top 'limit_comparisons' pairs
368             top_pairs = sorted_pairs[:limit_comparisons]
369
370             # Create new blocked_pairs based on top similarities and
371             existing blocked pairs
372             new_blocked_pairs = []
373             remaining_limit = limit_comparisons - len(blocked_pairs)
374
375             # First, include all existing blocked pairs
376             final_blocked_pairs = blocked_pairs.copy()
377
378             # Then, add new pairs from top similarities until we
379             reach the limit
380             for i, j in top_pairs:
381                 if remaining_limit <= 0:
382                     break
383                 left_item, right_item = left_data[i], right_data[j]
384                 left_key = get_hashable_key(left_item)
385                 right_key = get_hashable_key(right_item)
386                 if (left_key, right_key) not in block_pair_set:
387                     new_blocked_pairs.append((left_item, right_item))
388                     block_pair_set.add((left_key, right_key))
389                     remaining_limit -= 1
390
391             final_blocked_pairs.extend(new_blocked_pairs)
392             blocked_pairs = final_blocked_pairs
393
394             self.console.log(
395                 f"Limited comparisons to top {limit_comparisons}-
396                 pairs, including {len(blocked_pairs) - len(new_blocked_pairs)} from code-
397                 based blocking and {len(new_blocked_pairs)} based on cosine similarity.
398                 Lowest cosine similarity included: {similarities[top_pairs[-1]]:.4f}"
399             )
400         else:
401             # Add new pairs to blocked_pairs
402             for i, j in above_threshold:
403                 left_item, right_item = left_data[i], right_data[j]
404                 left_key = get_hashable_key(left_item)
405                 right_key = get_hashable_key(right_item)
406                 if (left_key, right_key) not in block_pair_set:
407                     blocked_pairs.append((left_item, right_item))
408                     block_pair_set.add((left_key, right_key))
409
410             # If there are no blocking conditions or embedding threshold, use
411             all pairs
412             if not blocking_conditions and blocking_threshold is None:
413                 blocked_pairs = [
414                     (left_item, right_item)
415                     for left_item in left_data
416                     for right_item in right_data
417                 ]

```

```

418         # If there's a limit on the number of comparisons, randomly
419         sample pairs
420         if limit_comparisons is not None and len(blocked_pairs) >
421             limit_comparisons:
422             self.console.log(
423                 f"Randomly sampling {limit_comparisons} pairs out of
424                 {len(blocked_pairs)} blocked pairs."
425             )
426             blocked_pairs = random.sample(blocked_pairs,
427                 limit_comparisons)
428
429             self.console.log(
430                 f"Total pairs to compare after blocking and sampling:
431                 {len(blocked_pairs)}"
432             )
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)),
471                     desc="Comparing pairs",
472                     console=self.console,
473                 )
474
475                 for i in pbar:
476                     future = list(future_to_pair.keys())[i]
477                     pair = future_to_pair[future]
478                     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.	<i>required</i>
model	str	The LLM model to use for comparison.	<i>required</i>
item1	dict	The first item to compare.	<i>required</i>
item2	dict	The second item to compare.	<i>required</i>
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`

```
92     def compare_pair(
93         self,
94         comparison_prompt: str,
95         model: str,
96         item1: dict,
97         item2: dict,
98         timeout_seconds: int = 120,
99         max_retries_per_timeout: int = 2,
100     ) -> tuple[bool, float]:
101         """
102             Compares two items using an LLM model to determine if they match.
103
104         Args:
105             comparison_prompt (str): The prompt template for comparison.
106             model (str): The LLM model to use for comparison.
107             item1 (dict): The first item to compare.
108             item2 (dict): The second item to compare.
109             timeout_seconds (int): The timeout for the LLM call in seconds.
110             max_retries_per_timeout (int): The maximum number of retries per
111             timeout.
112
113         Returns:
114             tuple[bool, float]: A tuple containing a boolean indicating
115             whether the items match and the cost of the comparison.
116             """
117
118         try:
119             prompt = strict_render(comparison_prompt, {"left": item1,
120 "right": item2})
121             except Exception as e:
122                 self.console.log(f"[red]Error rendering prompt: {e}[/red]")
123                 return False, 0
124             response = self.runner.api.call_llm(
125                 model,
126                 "compare",
127                 [{"role": "user", "content": prompt}],
128                 {"is_match": "bool"},
129                 timeout_seconds=timeout_seconds,
130                 max_retries_per_timeout=max_retries_per_timeout,
131                 bypass_cache=self.config.get("bypass_cache", self.bypass_cache),
132
133             litellm_completion_kwargs=self.config.get("litellm_completion_kwargs",
134             {}),
135                 op_config=self.config,
136             )
137             cost = 0
138             try:
139                 cost = response.total_cost
140                 output = self.runner.api.parse_llm_response(
141                     response.response, {"is_match": "bool"}
142                 )[0]
143             except Exception as e:
144                 self.console.log(f"[red]Error parsing LLM response: {e}[/red]")
145                 return False, cost
146             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.	<i>required</i>
right_data	list[dict]	The right dataset to join.	<i>required</i>

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`

```

143     def execute(
144         self, left_data: list[dict], right_data: list[dict]
145     ) -> tuple[list[dict], float]:
146         """
147             Executes the equijoin operation on the provided datasets.
148
149             Args:
150                 left_data (list[dict]): The left dataset to join.
151                 right_data (list[dict]): The right dataset to join.
152
153             Returns:
154                 tuple[list[dict], float]: A tuple containing the joined results
155                 and the total cost of the operation.
156
157             Usage:
158             ```python
159             from docetl.operations import EquijoinOperation
160
161             config = {
162                 "blocking_keys": {
163                     "left": ["id"],
164                     "right": ["user_id"]
165                 },
166                 "limits": {
167                     "left": 1,
168                     "right": 1
169                 },
170                 "comparison_prompt": "Compare {{left}} and {{right}} and
determine if they match.",
171                 "blocking_threshold": 0.8,
172                 "blocking_conditions": ["left['id'] == right['user_id']"],
173                 "limit_comparisons": 1000
174             }
175             equijoin_op = EquijoinOperation(config)
176             left_data = [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]
177             right_data = [{"user_id": 1, "age": 30}, {"user_id": 2, "age": 25}]
178             results, cost = equijoin_op.execute(left_data, right_data)
179             print(f"Joined results: {results}")
180             print(f"Total cost: {cost}")
181             ```
182
183
184             This method performs the following steps:
185             1. Initial blocking based on specified conditions (if any)
186             2. Embedding-based blocking (if threshold is provided)
187             3. LLM-based comparison for blocked pairs
188             4. Result aggregation and validation
189
190             The method also calculates and logs statistics such as comparisons
191             saved by blocking and join selectivity.
192             """
193
194             blocking_keys = self.config.get("blocking_keys", {})
195             left_keys = blocking_keys.get(
196                 "left", list(left_data[0].keys()) if left_data else []
197             )
198             right_keys = blocking_keys.get(
199                 "right", list(right_data[0].keys()) if right_data else []

```

```

200     )
201     limits = self.config.get(
202         "limits", {"left": float("inf"), "right": float("inf")})
203     )
204     left_limit = limits["left"]
205     right_limit = limits["right"]
206     blocking_threshold = self.config.get("blocking_threshold")
207     blocking_conditions = self.config.get("blocking_conditions", [])
208     limit_comparisons = self.config.get("limit_comparisons")
209     total_cost = 0
210
211     if len(left_data) == 0 or len(right_data) == 0:
212         return [], 0
213
214     if self.status:
215         self.status.stop()
216
217     # Initial blocking using multiprocessing
218     num_processes = min(cpu_count(), len(left_data))
219
220     self.console.log(
221         f"Starting to run code-based blocking rules for {len(left_data)}"
222         f"left and {len(right_data)} right rows ({len(left_data)} * {len(right_data)}"
223         f"total pairs) with {num_processes} processes..."
224     )
225
226     with Pool(
227         processes=num_processes,
228         initializer=init_worker,
229         initargs=(right_data, blocking_conditions),
230     ) as pool:
231         blocked_pairs_nested = pool.map(process_left_item, left_data)
232
233         # Flatten the nested list of blocked pairs
234         blocked_pairs = [pair for sublist in blocked_pairs_nested for pair in
235                         sublist]
236
237         # Check if we have exceeded the pairwise comparison limit
238         if limit_comparisons is not None and len(blocked_pairs) >
239             limit_comparisons:
240             # Sample pairs based on cardinality and length
241             sampled_pairs = stratified_length_sample(
242                 blocked_pairs, limit_comparisons, sample_size=1000,
243                 console=self.console
244             )
245
246             # Calculate number of dropped pairs
247             dropped_pairs = len(blocked_pairs) - limit_comparisons
248
249             # Prompt the user for confirmation
250             if self.status:
251                 self.status.stop()
252                 if not Confirm.ask(
253                     f"[yellow]Warning: {dropped_pairs} pairs will be dropped due"
254                     f"to the comparison limit. "
255                     f"Proceeding with {limit_comparisons} randomly sampled pairs."
256                 ):
257                     f"Do you want to continue?[/yellow]",
258                     console=self.console,
259                 ):
260                     raise ValueError("Operation cancelled by user due to pair

```

```

261     limit.")
262
263     if self.status:
264         self.status.start()
265
266     blocked_pairs = sampled_pairs
267
268     self.console.log(
269         f"Number of blocked pairs after initial blocking:
270 {len(blocked_pairs)}"
271     )
272
273     if blocking_threshold is not None:
274         embedding_model = self.config.get("embedding_model",
275         self.default_model)
276         model_input_context_length = model_cost.get(embedding_model,
277         {}).get(
278             "max_input_tokens", 8192
279         )
280
281     def get_embeddings(
282         input_data: list[dict[str, Any]], keys: list[str], name: str
283     ) -> tuple[list[list[float]], float]:
284         texts = [
285             " ".join(str(item[key])) for key in keys if key in item>[
286                 : model_input_context_length * 4
287             ]
288             for item in input_data
289         ]
290
291         embeddings = []
292         total_cost = 0
293         batch_size = 2000
294         for i in range(0, len(texts), batch_size):
295             batch = texts[i : i + batch_size]
296             self.console.log(
297                 f"On iteration {i} for creating embeddings for {name}"
298             )
299             )
300             response = self.runner.api.gen_embedding(
301                 model=embedding_model,
302                 input=batch,
303                 )
304             embeddings.extend([data["embedding"] for data in
305             response["data"]])
306             total_cost += completion_cost(response)
307             return embeddings, total_cost
308
309             left_embeddings, left_cost = get_embeddings(left_data, left_keys,
310             "left")
311             right_embeddings, right_cost = get_embeddings(
312                 right_data, right_keys, "right"
313             )
314             total_cost += left_cost + right_cost
315             self.console.log(
316                 f"Created embeddings for datasets. Total embedding creation
317             cost: {total_cost}"
318             )
319
320             # Compute all cosine similarities in one call
321             from sklearn.metrics.pairwise import cosine_similarity

```

```

322         similarities = cosine_similarity(left_embeddings,
323     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}"

```

```

383         )
384     else:
385         # Add new pairs to blocked_pairs
386         for i, j in above_threshold:
387             left_item, right_item = left_data[i], right_data[j]
388             left_key = get_hashable_key(left_item)
389             right_key = get_hashable_key(right_item)
390             if (left_key, right_key) not in block_pair_set:
391                 blocked_pairs.append((left_item, right_item))
392                 block_pair_set.add((left_key, right_key))
393
394         # If there are no blocking conditions or embedding threshold, use all
395         pairs
396         if not blocking_conditions and blocking_threshold is None:
397             blocked_pairs = [
398                 (left_item, right_item)
399                 for left_item in left_data
400                 for right_item in right_data
401             ]
402
403         # If there's a limit on the number of comparisons, randomly sample
404         pairs
405         if limit_comparisons is not None and len(blocked_pairs) >
406             limit_comparisons:
407             self.console.log(
408                 f"Randomly sampling {limit_comparisons} pairs out of
409                 {len(blocked_pairs)} blocked pairs."
410             )
411             blocked_pairs = random.sample(blocked_pairs, limit_comparisons)
412
413             self.console.log(
414                 f"Total pairs to compare after blocking and sampling:
415                 {len(blocked_pairs)}"
416             )
417
418         # Calculate and print statistics
419         total_possible_comparisons = len(left_data) * len(right_data)
420         comparisons_made = len(blocked_pairs)
421         comparisons_saved = total_possible_comparisons - comparisons_made
422         self.console.log(
423             f"[green]Comparisons saved by blocking: {comparisons_saved} "
424             f"({(comparisons_saved / total_possible_comparisons) * 100:.2f}%)
425             [/green]"
426         )
427
428         left_match_counts = defaultdict(int)
429         right_match_counts = defaultdict(int)
430         results = []
431         comparison_costs = 0
432
433         if self.status:
434             self.status.stop()
435
436         with ThreadPoolExecutor(max_workers=self.max_threads) as executor:
437             future_to_pair = {
438                 executor.submit(
439                     self.compare_pair,
440                     self.config["comparison_prompt"],
441                     self.config.get("comparison_model", self.default_model),
442                     left,
443                     right,

```

```

444         self.config.get("timeout", 120),
445         self.config.get("max_retries_per_timeout", 2),
446     ): (left, right)
447     for left, right in blocked_pairs
448     ):
449
450     pbar = RichLoopBar(
451         range(len(future_to_pair)),
452         desc="Comparing pairs",
453         console=self.console,
454     )
455
456     for i in pbar:
457         future = list(future_to_pair.keys())[i]
458         pair = future_to_pair[future]
459         is_match, cost = future.result()
460         comparison_costs += cost
461
462         if is_match:
463             joined_item = {}
464             left_item, right_item = pair
465             left_key_hash = get_hashable_key(left_item)
466             right_key_hash = get_hashable_key(right_item)
467             if (
468                 left_match_counts[left_key_hash] >= left_limit
469                 or right_match_counts[right_key_hash] >= right_limit
470             ):
471                 continue
472
473                 for key, value in left_item.items():
474                     joined_item[f"{key}_left" if key in right_item else
475 key] = value
476                     for key, value in right_item.items():
477                         joined_item[f"{key}_right" if key in left_item else
478 key] = value
479                         if self.runner.api.validate_output(
480                             self.config, joined_item, self.console
481                         ):
482                             results.append(joined_item)
483                             left_match_counts[left_key_hash] += 1
484                             right_match_counts[right_key_hash] += 1
485
486                         # TODO: support retry in validation failure
487
488             total_cost += comparison_costs
489
490             if self.status:
491                 self.status.start()
492
493             # Calculate and print the join selectivity
494             join_selectivity = (
495                 len(results) / (len(left_data) * len(right_data))
496                 if len(left_data) * len(right_data) > 0
497                 else 0
498             )
499             self.console.log(f"Equijoin selectivity: {join_selectivity:.4f}")
500
501             if self.status:
502                 self.status.start()

```

```
    return results, total_cost
```

`docetl.operations.cluster.ClusterOperation`

Bases: `BaseOperation`

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

```
12  class ClusterOperation(BaseOperation):
13      def __init__(
14          self,
15          *args,
16          **kwargs,
17      ):
18          super().__init__(*args, **kwargs)
19          self.max_batch_size: int = self.config.get(
20              "max_batch_size", kwargs.get("max_batch_size", float("inf"))
21          )
22
23      def syntax_check(self) -> None:
24          """
25              Checks the configuration of the ClusterOperation for required
26              keys and valid structure.
27
28          Raises:
29              ValueError: If required keys are missing or invalid in the
30              configuration.
31              TypeError: If configuration values have incorrect types.
32          """
33          required_keys = ["embedding_keys", "summary_schema",
34 "summary_prompt"]
35          for key in required_keys:
36              if key not in self.config:
37                  raise ValueError(
38                      f"Missing required key '{key}' in ClusterOperation
configuration"
39                  )
40
41          if not isinstance(self.config["embedding_keys"], list):
42              raise TypeError("'embedding_keys' must be a list of strings")
43
44          if "output_key" in self.config:
45              if not isinstance(self.config["output_key"], str):
46                  raise TypeError("'output_key' must be a string")
47
48          if not isinstance(self.config["summary_schema"], dict):
49              raise TypeError("'summary_schema' must be a dictionary")
50
51          if not isinstance(self.config["summary_prompt"], str):
52              raise TypeError("'prompt' must be a string")
53
54          # Check if the prompt is a valid Jinja2 template
55          try:
56              Template(self.config["summary_prompt"])
57          except Exception as e:
58              raise ValueError(f"Invalid Jinja2 template in 'prompt':
{str(e)}")
59
60          # Check optional parameters
61          if "max_batch_size" in self.config:
62              if not isinstance(self.config["max_batch_size"], int):
63                  raise TypeError("'max_batch_size' must be an integer")
64
65          if "embedding_model" in self.config:
66              if not isinstance(self.config["embedding_model"], str):
```

```

69             raise TypeError("'embedding_model' must be a string")
70
71     if "model" in self.config:
72         if not isinstance(self.config["model"], str):
73             raise TypeError("'model' must be a string")
74
75     if "validate" in self.config:
76         if not isinstance(self.config["validate"], list):
77             raise TypeError("'validate' must be a list of strings")
78         for rule in self.config["validate"]:
79             if not isinstance(rule, str):
80                 raise TypeError("Each validation rule must be a
81 string")
82
83     def execute(
84         self, input_data: list[dict], is_build: bool = False
85     ) -> tuple[list[dict], float]:
86         """
87             Executes the cluster operation on the input data. Modifies the
88             input data and returns it in place.
89
90         Args:
91             input_data (list[dict]): A list of dictionaries to process.
92             is_build (bool): Whether the operation is being executed
93                 in the build phase. Defaults to False.
94
95         Returns:
96             tuple[list[dict], float]: A tuple containing the clustered
97                 list of dictionaries and the total cost of the operation.
98         """
99         if not input_data:
100             return input_data, 0
101
102         if len(input_data) == 1:
103             input_data[0][self.config.get("output_key", "clusters")] = ()
104             return input_data, 0
105
106         embeddings, cost = get_embeddings_for_clustering(
107             input_data, self.config, self.runner.api
108         )
109
110         tree = self.agglomerative_cluster_of_embeddings(input_data,
111         embeddings)
112
113         if "collapse" in self.config:
114             tree = self.collapse_tree(tree,
115             collapse=self.config["collapse"])
116
117             self.prompt_template = Template(self.config["summary_prompt"])
118             cost += self.annotate_clustering_tree(tree)
119             self.annotate_leaves(tree)
120
121             return input_data, cost
122
123     def agglomerative_cluster_of_embeddings(self, input_data,
124     embeddings):
125         import sklearn.cluster
126
127         cl = sklearn.cluster.AgglomerativeClustering(
128             compute_full_tree=True, compute_distances=True
129         )

```

```
130         cl.fit(embeddings)
131
132     nsamples = len(embeddings)
133
134     def build_tree(i):
135         if i < nsamples:
136             res = input_data[i]
137             #                         res["embedding"] = list(embeddings[i])
138             return res
139         return {
140             "children": [
141                 build_tree(cl.children_[i - nsamples, 0]),
142                 build_tree(cl.children_[i - nsamples, 1]),
143             ],
144             "distance": cl.distances_[i - nsamples],
145         }
146
147     return build_tree(nsamples + len(cl.children_) - 1)
148
149 def get_tree_distances(self, t):
150     res = set()
151     if "distance" in t:
152         res.update(
153             set(
154                 [
155                     t["distance"] - child["distance"]
156                     for child in t["children"]
157                     if "distance" in child
158                 ]
159             )
160         )
161     if "children" in t:
162         for child in t["children"]:
163             res.update(self.get_tree_distances(child))
164     return res
165
166 def _collapse_tree(self, t, parent_dist=None, collapse=None):
167     if "children" in t:
168         if (
169             "distance" in t
170             and parent_dist is not None
171             and collapse is not None
172             and parent_dist - t["distance"] < collapse
173         ):
174             return [
175                 grandchild
176                 for child in t["children"]
177                 for grandchild in self._collapse_tree(
178                     child, parent_dist=parent_dist, collapse=collapse
179                 )
180             ]
181         else:
182             res = dict(t)
183             res["children"] = [
184                 grandchild
185                 for idx, child in enumerate(t["children"])
186                 for grandchild in self._collapse_tree(
187                     child, parent_dist=t["distance"],
188                     collapse=collapse
189                 )
190             ]

```

```

191         return [res]
192     else:
193         return [t]
194
195     def collapse_tree(self, tree, collapse=None):
196         if collapse is not None:
197             tree_distances =
198             np.array(sorted(self.get_tree_distances(tree)))
199             collapse = tree_distances[int(len(tree_distances) * *
200 collapse)]
201             return self._collapse_tree(tree, collapse=collapse)[0]
202
203     def annotate_clustering_tree(self, t):
204         if "children" in t:
205             with ThreadPoolExecutor(max_workers=self.max_batch_size) as
206             executor:
207                 futures = [
208                     executor.submit(self.annotate_clustering_tree, child)
209                     for child in t["children"]
210                 ]
211
212                 total_cost = 0
213                 pbar = RichLoopBar(
214                     range(len(futures)),
215                     desc=f"Processing {self.config['name']} (map) on all
216 documents",
217                     console=self.console,
218                 )
219                 for i in pbar:
220                     total_cost += futures[i].result()
221                     pbar.update(i)
222
223                 prompt = strict_render(self.prompt_template, {"inputs":*
224 t["children"]})
225
226                 def validation_fn(response: dict[str, Any]):
227                     output = self.runner.api.parse_llm_response(
228                         response,
229                         schema=self.config["summary_schema"],
230                         manually_fix_errors=self.manually_fix_errors,
231                     )[0]
232                     if self.runner.api.validate_output(self.config, output,
233 self.console):
234                         return output, True
235                         return output, False
236
237                     response = self.runner.api.call_llm(
238                         model=self.config.get("model", self.default_model),
239                         op_type="cluster",
240                         messages=[{"role": "user", "content": prompt}],
241                         output_schema=self.config["summary_schema"],
242                         timeout_seconds=self.config.get("timeout", 120),
243                         bypass_cache=self.config.get("bypass_cache",
244 self.bypass_cache),
245
246                         max_retries_per_timeout=self.config.get("max_retries_per_timeout", 2),
247                         validation_config=(
248                             {
249                                 "num_retries":*
250 self.num_retries_on_validate_failure,
251                                 "val_rule": self.config.get("validate", []),
252

```

```

252                     "validation_fn": validation_fn,
253                 }
254             if self.config.get("validate", None)
255             else None
256         ),
257         verbose=self.config.get("verbose", False),
258         litellm_completion_kwargs=self.config.get(
259             "litellm_completion_kwargs", {}
260         ),
261         op_config=self.config,
262     )
263     total_cost += response.total_cost
264     if response.validated:
265         output = self.runner.api.parse_llm_response(
266             response.response,
267             schema=self.config["summary_schema"],
268             manually_fix_errors=self.manually_fix_errors,
269         )[0]
270         t.update(output)

271     return total_cost
272 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.	<i>required</i>
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`

```

77     def execute(
78         self, input_data: list[dict], is_build: bool = False
79     ) -> tuple[list[dict], float]:
80         """
81             Executes the cluster operation on the input data. Modifies the
82             input data and returns it in place.
83
84         Args:
85             input_data (list[dict]): A list of dictionaries to process.
86             is_build (bool): Whether the operation is being executed
87                 in the build phase. Defaults to False.
88
89         Returns:
90             tuple[list[dict], float]: A tuple containing the clustered
91                 list of dictionaries and the total cost of the operation.
92         """
93         if not input_data:
94             return input_data, 0
95
96         if len(input_data) == 1:
97             input_data[0][self.config.get("output_key", "clusters")] = ()
98             return input_data, 0
99
100        embeddings, cost = get_embeddings_for_clustering(
101            input_data, self.config, self.runner.api
102        )
103
104        tree = self.agglomerative_cluster_of_embeddings(input_data,
105        embeddings)
106
107        if "collapse" in self.config:
108            tree = self.collapse_tree(tree, collapse=self.config["collapse"])
109
110        self.prompt_template = Template(self.config["summary_prompt"])
111        cost += self.annotate_clustering_tree(tree)
112        self.annotate_leaves(tree)
113
114        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`

```
23 def syntax_check(self) -> None:
24     """
25         Checks the configuration of the ClusterOperation for required keys and
26         valid structure.
27
28         Raises:
29             ValueError: If required keys are missing or invalid in the
30             configuration.
31             TypeError: If configuration values have incorrect types.
32         """
33     required_keys = ["embedding_keys", "summary_schema", "summary_prompt"]
34     for key in required_keys:
35         if key not in self.config:
36             raise ValueError(
37                 f"Missing required key '{key}' in ClusterOperation
38             configuration"
39             )
40
41     if not isinstance(self.config["embedding_keys"], list):
42         raise TypeError("'embedding_keys' must be a list of strings")
43
44     if "output_key" in self.config:
45         if not isinstance(self.config["output_key"], str):
46             raise TypeError("'output_key' must be a string")
47
48     if not isinstance(self.config["summary_schema"], dict):
49         raise TypeError("'summary_schema' must be a dictionary")
50
51     if not isinstance(self.config["summary_prompt"], str):
52         raise TypeError("'prompt' must be a string")
53
54     # Check if the prompt is a valid Jinja2 template
55     try:
56         Template(self.config["summary_prompt"])
57     except Exception as e:
58         raise ValueError(f"Invalid Jinja2 template in 'prompt': {str(e)}")
59
60     # Check optional parameters
61     if "max_batch_size" in self.config:
62         if not isinstance(self.config["max_batch_size"], int):
63             raise TypeError("'max_batch_size' must be an integer")
64
65     if "embedding_model" in self.config:
66         if not isinstance(self.config["embedding_model"], str):
67             raise TypeError("'embedding_model' must be a string")
68
69     if "model" in self.config:
70         if not isinstance(self.config["model"], str):
71             raise TypeError("'model' must be a string")
72
73     if "validate" in self.config:
74         if not isinstance(self.config["validate"], list):
75             raise TypeError("'validate' must be a list of strings")
76         for rule in self.config["validate"]:
77             if not isinstance(rule, str):
78                 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`

```
10  class SplitOperation(BaseOperation):
11      """
12          A class that implements a split operation on input data, dividing it
13          into manageable chunks.
14
15          This class extends BaseOperation to:
16              1. Split input data into chunks of specified size based on the
17                  'split_key' and 'token_count' configuration.
18              2. Assign unique identifiers to each original document and number
19                  chunks sequentially.
20              3. Return results containing:
21                  - {split_key}_chunk: The content of the split chunk.
22                  - {name}_id: A unique identifier for each original document.
23                  - {name}_chunk_num: The sequential number of the chunk within its
24                      original document.
25      """
26
27  class schema(BaseOperation.schema):
28      type: str = "split"
29      split_key: str
30      method: str
31      method_kwargs: dict[str, Any]
32      model: str | None = None
33
34      @field_validator("method")
35      def validate_method(cls, v):
36          if v not in ["token_count", "delimiter"]:
37              raise ValueError(
38                  f"Invalid method '{v}'. Must be 'token_count' or
39                  'delimiter'"
40              )
41          return v
42
43      @model_validator(mode="after")
44      def validate_method_kwargs(self):
45          if self.method == "token_count":
46              num_tokens = self.method_kwargs.get("num_tokens")
47              if num_tokens is None or num_tokens <= 0:
48                  raise ValueError("'num_tokens' must be a positive
49                  integer")
50          elif self.method == "delimiter":
51              if "delimiter" not in self.method_kwargs:
52                  raise ValueError("'delimiter' is required for
53                  delimiter method")
54          return self
55
56      def __init__(self, *args, **kwargs):
57          super().__init__(*args, **kwargs)
58          self.name = self.config["name"]
59
60      def execute(self, input_data: list[dict]) -> tuple[list[dict],
61 float]:
62          split_key = self.config["split_key"]
63          method = self.config["method"]
64          method_kwargs = self.config["method_kwargs"]
65          try:
66              encoder = tiktoken.encoding_for_model(
```

```
67         self.config["method_kwargs"]
68             .get("model", self.default_model)
69             .split("/")[-1]
70     )
71 except Exception:
72     encoder = tiktoken.encoding_for_model("gpt-4o")
73
74 results = []
75 cost = 0.0
76
77 for item in input_data:
78     if split_key not in item:
79         raise KeyError(f"Split key '{split_key}' not found in
80 item")
81
82     content = item[split_key]
83     doc_id = str(uuid.uuid4())
84
85     if method == "token_count":
86         token_count = method_kwargs["num_tokens"]
87         tokens = encoder.encode(content)
88
89         for chunk_num, i in enumerate(
90             range(0, len(tokens), token_count), start=1
91         ):
92             chunk_tokens = tokens[i : i + token_count]
93             chunk = encoder.decode(chunk_tokens)
94
95             result = item.copy()
96             result.update(
97                 {
98                     f"{split_key}_chunk": chunk,
99                     f"{self.name}_id": doc_id,
100                    f"{self.name}_chunk_num": chunk_num,
101                }
102            )
103             results.append(result)
104
105 elif method == "delimiter":
106     delimiter = method_kwargs["delimiter"]
107     num_splits_to_group =
108 method_kwargs.get("num_splits_to_group", 1)
109     chunks = content.split(delimiter)
110
111     # Get rid of empty chunks
112     chunks = [chunk for chunk in chunks if chunk.strip()]
113
114     for chunk_num, i in enumerate(
115         range(0, len(chunks), num_splits_to_group), start=1
116     ):
117         grouped_chunks = chunks[i : i + num_splits_to_group]
118         joined_chunk = delimiter.join(grouped_chunks).strip()
119
120         result = item.copy()
121         result.update(
122             {
123                 f"{split_key}_chunk": joined_chunk,
124                 f"{self.name}_id": doc_id,
125                 f"{self.name}_chunk_num": chunk_num,
126             }
127         )
128     )
```

```
        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`

```
8  class GatherOperation(BaseOperation):
9      """
10         A class that implements a gather operation on input data, adding
11         contextual information from surrounding chunks.
12
13         This class extends BaseOperation to:
14             1. Group chunks by their document ID.
15             2. Order chunks within each group.
16             3. Add peripheral context to each chunk based on the configuration.
17             4. Include headers for each chunk and its upward hierarchy.
18             5. Return results containing the rendered chunks with added context,
19         including information about skipped characters and headers.
20         """
21
22     class schema(BaseOperation.schema):
23         type: str = "gather"
24         content_key: str
25         doc_id_key: str
26         order_key: str
27         peripheral_chunks: dict[str, Any] | None = None
28         doc_header_key: str | None = None
29         main_chunk_start: str | None = None
30         main_chunk_end: str | None = None
31
32         @field_validator("peripheral_chunks")
33         def validate_peripheral_chunks(cls, v):
34             for direction in ["previous", "next"]:
35                 if direction not in v:
36                     continue
37                 for section in ["head", "middle", "tail"]:
38                     if section in v[direction]:
39                         section_config = v[direction][section]
40                         if section != "middle" and "count" not in
41                         section_config:
42                             raise ValueError(
43                                 f"Missing 'count' in {direction}.
44 {section} configuration"
45                             )
46             return v
47
48     def __init__(self, *args: Any, **kwargs: Any) -> None:
49         """
50             Initialize the GatherOperation.
51
52             Args:
53                 *args: Variable length argument list.
54                 **kwargs: Arbitrary keyword arguments.
55             """
56             super().__init__(*args, **kwargs)
57
58     def syntax_check(self) -> None:
59         """Perform a syntax check on the operation configuration."""
60         # Validate the schema using Pydantic
61         self.schema(**self.config)
62
63         def execute(self, input_data: list[dict]) -> tuple[list[dict],
64         float]:
```

```
65     """
66     Execute the gather operation on the input data.
67
68     Args:
69         input_data (list[dict]): The input data to process.
70
71     Returns:
72         tuple[list[dict], float]: A tuple containing the processed
73         results and the cost of the operation.
74     """
75     content_key = self.config["content_key"]
76     doc_id_key = self.config["doc_id_key"]
77     order_key = self.config["order_key"]
78     peripheral_config = self.config.get("peripheral_chunks", {})
79     main_chunk_start = self.config.get(
80         "main_chunk_start", "---- Begin Main Chunk ---"
81     )
82     main_chunk_end = self.config.get("main_chunk_end", "--- End Main
83     Chunk ---")
84     doc_header_key = self.config.get("doc_header_key", None)
85     results = []
86     cost = 0.0
87
88     # Group chunks by document ID
89     grouped_chunks = {}
90     for item in input_data:
91         doc_id = item[doc_id_key]
92         if doc_id not in grouped_chunks:
93             grouped_chunks[doc_id] = []
94             grouped_chunks[doc_id].append(item)
95
96     # Process each group of chunks
97     for chunks in grouped_chunks.values():
98         # Sort chunks by their order within the document
99         chunks.sort(key=lambda x: x[order_key])
100
101     # Process each chunk with its peripheral context and headers
102     for i, chunk in enumerate(chunks):
103         rendered_chunk = self.render_chunk_with_context(
104             chunks,
105             i,
106             peripheral_config,
107             content_key,
108             order_key,
109             main_chunk_start,
110             main_chunk_end,
111             doc_header_key,
112         )
113
114         result = chunk.copy()
115         result[f"{content_key}_rendered"] = rendered_chunk
116         results.append(result)
117
118     return results, cost
119
120     def render_chunk_with_context(
121         self,
122         chunks: list[dict],
123         current_index: int,
124         peripheral_config: dict,
125         content_key: str,
```

```

126     order_key: str,
127     main_chunk_start: str,
128     main_chunk_end: str,
129     doc_header_key: str,
130 ) -> str:
131     """
132     Render a chunk with its peripheral context and headers.
133
134     Args:
135         chunks (list[dict]): List of all chunks in the document.
136         current_index (int): Index of the current chunk being
137         processed.
138         peripheral_config (dict): Configuration for peripheral
139         chunks.
140         content_key (str): Key for the content in each chunk.
141         order_key (str): Key for the order of each chunk.
142         main_chunk_start (str): String to mark the start of the main
143         chunk.
144         main_chunk_end (str): String to mark the end of the main
145         chunk.
146         doc_header_key (str): The key for the headers in the current
147         chunk.
148
149     Returns:
150         str: Rendered chunk with context and headers.
151     """
152
153     # If there are no peripheral chunks, return the main chunk
154     if not peripheral_config:
155         return chunks[current_index][content_key]
156
157     combined_parts = ["--- Previous Context ---"]
158
159     combined_parts.extend(
160         self.process_peripheral_chunks(
161             chunks[:current_index],
162             peripheral_config.get("previous", {}),
163             content_key,
164             order_key,
165         )
166     )
167     combined_parts.append("--- End Previous Context ---\n")
168
169     # Process main chunk
170     main_chunk = chunks[current_index]
171     if headers := self.render_hierarchy_headers(
172         main_chunk, chunks[: current_index + 1], doc_header_key
173     ):
174         combined_parts.append(headers)
175     combined_parts.extend(
176         (
177             f"{main_chunk_start}",
178             f"{main_chunk[content_key]}",
179             f"{main_chunk_end}",
180             "\n--- Next Context ---",
181         )
182     )
183     combined_parts.extend(
184         self.process_peripheral_chunks(
185             chunks[current_index + 1 :],
186             peripheral_config.get("next", {}),
187         ),

```

```
187         content_key,
188         order_key,
189     )
190 )
191 combined_parts.append("--- End Next Context ---")
192
193 return "\n".join(combined_parts)
194
195 def process_peripheral_chunks(
196     self,
197     chunks: list[dict],
198     config: dict,
199     content_key: str,
200     order_key: str,
201     reverse: bool = False,
202 ) -> list[str]:
203 """
204     Process peripheral chunks according to the configuration.
205
206     Args:
207         chunks (list[dict]): List of chunks to process.
208         config (dict): Configuration for processing peripheral
209         chunks.
210         content_key (str): Key for the content in each chunk.
211         order_key (str): Key for the order of each chunk.
212         reverse (bool, optional): Whether to process chunks in
213         reverse order. Defaults to False.
214
215     Returns:
216         list[str]: List of processed chunk strings.
217 """
218     if reverse:
219         chunks = list(reversed(chunks))
220
221     processed_parts = []
222     included_chunks = []
223     total_chunks = len(chunks)
224
225     head_config = config.get("head", {})
226     tail_config = config.get("tail", {})
227
228     head_count = int(head_config.get("count", 0))
229     tail_count = int(tail_config.get("count", 0))
230     in_skip = False
231     skip_char_count = 0
232
233     for i, chunk in enumerate(chunks):
234         if i < head_count:
235             section = "head"
236         elif i >= total_chunks - tail_count:
237             section = "tail"
238         elif "middle" in config:
239             section = "middle"
240         else:
241             # Show number of characters skipped
242             skipped_chars = len(chunk[content_key])
243             if not in_skip:
244                 skip_char_count = skipped_chars
245                 in_skip = True
246             else:
247                 skip_char_count += skipped_chars
```

```

248         continue
249
250
251     if in_skip:
252         processed_parts.append(
253             f"[... {skip_char_count} characters skipped ...]"
254         )
255         in_skip = False
256         skip_char_count = 0
257
258         section_config = config.get(section, {})
259         section_content_key = section_config.get("content_key",
260         content_key)
261
262         is_summary = section_content_key != content_key
263         summary_suffix = " (Summary)" if is_summary else ""
264
265         chunk_prefix = f"[Chunk {chunk[order_key]}{summary_suffix}]"
266         processed_parts.extend((chunk_prefix, f"
267         {chunk[section_content_key]}"))
268         included_chunks.append(chunk)
269
270     if in_skip:
271         processed_parts.append(f"[... {skip_char_count} characters
272 skipped ...]")
273
274     if reverse:
275         processed_parts = list(reversed(processed_parts))
276
277     return processed_parts
278
279 def render_hierarchy_headers(
280     self,
281     current_chunk: dict,
282     chunks: list[dict],
283     doc_header_key: str,
284 ) -> str:
285     """
286     Render headers for the current chunk's hierarchy.
287
288     Args:
289         current_chunk (dict): The current chunk being processed.
290         chunks (list[dict]): List of chunks up to and including the
291     current chunk.
292         doc_header_key (str): The key for the headers in the current
293     chunk.
294     Returns:
295         str: Rendered headers in the current chunk's hierarchy.
296     """
297     current_hierarchy = {}
298
299     if doc_header_key is None:
300         return ""
301
302     # Find the largest/highest level in the current chunk
303     current_chunk_headers = current_chunk.get(doc_header_key, [])
304
305     # If there are no headers in the current chunk, return an empty
306     string
307     if not current_chunk_headers:
308         return ""

```

```

309     highest_level = float("inf") # Initialize with positive infinity
310     for header_info in current_chunk_headers:
311         try:
312             level = header_info.get("level")
313             if level is not None and level < highest_level:
314                 highest_level = level
315             except Exception as e:
316                 self.runner.console.log(f"[red]Error processing header: {e}[/red]")
317                 self.runner.console.log(f"[red]Header: {header_info}[/red]")
318             return ""
319
320     # If no headers found in the current chunk, set highest_level to
321     None
322     if highest_level == float("inf"):
323         highest_level = None
324
325     for chunk in chunks:
326         for header_info in chunk.get(doc_header_key, []):
327             try:
328                 header = header_info["header"]
329                 level = header_info["level"]
330                 if header and level:
331                     current_hierarchy[level] = header
332                     # Clear lower levels when a higher level header is
333                     found
334                     for lower_level in range(level + 1,
335 len(current_hierarchy) + 1):
336                         if lower_level in current_hierarchy:
337                             current_hierarchy[lower_level] = None
338             except Exception as e:
339                 self.runner.console.log(f"[red]Error processing
340 header: {e}[/red]")
341                 self.runner.console.log(f"[red]Header: {header_info}[/red]")
342             return ""
343
344             rendered_headers = [
345                 f"{'#' * level} {header}"
346                 for level, header in sorted(current_hierarchy.items())
347                 if header is not None and (highest_level is None or level <
348 highest_level)
349             ]
350             rendered_headers = " > ".join(rendered_headers)
351             return f"_Current Section:_ {rendered_headers}" if
352 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.	<i>required</i>

Returns:

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

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

```
59  def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
60      """
61      Execute the gather operation on the input data.
62
63      Args:
64          input_data (list[dict]): The input data to process.
65
66      Returns:
67          tuple[list[dict], float]: A tuple containing the processed
68      results and the cost of the operation.
69      """
70      content_key = self.config["content_key"]
71      doc_id_key = self.config["doc_id_key"]
72      order_key = self.config["order_key"]
73      peripheral_config = self.config.get("peripheral_chunks", {})
74      main_chunk_start = self.config.get(
75          "main_chunk_start", "--- Begin Main Chunk ---"
76      )
77      main_chunk_end = self.config.get("main_chunk_end", "--- End Main
78      Chunk ---")
79      doc_header_key = self.config.get("doc_header_key", None)
80      results = []
81      cost = 0.0
82
83      # Group chunks by document ID
84      grouped_chunks = {}
85      for item in input_data:
86          doc_id = item[doc_id_key]
87          if doc_id not in grouped_chunks:
88              grouped_chunks[doc_id] = []
89              grouped_chunks[doc_id].append(item)
90
91      # Process each group of chunks
92      for chunks in grouped_chunks.values():
93          # Sort chunks by their order within the document
94          chunks.sort(key=lambda x: x[order_key])
95
96          # Process each chunk with its peripheral context and headers
97          for i, chunk in enumerate(chunks):
98              rendered_chunk = self.render_chunk_with_context(
99                  chunks,
100                  i,
101                  peripheral_config,
102                  content_key,
103                  order_key,
104                  main_chunk_start,
105                  main_chunk_end,
106                  doc_header_key,
107              )
108
109              result = chunk.copy()
110              result[f"{content_key}_rendered"] = rendered_chunk
111              results.append(result)
112
113      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`

```
183     def process_peripheral_chunks(
184         self,
185         chunks: list[dict],
186         config: dict,
187         content_key: str,
188         order_key: str,
189         reverse: bool = False,
190     ) -> list[str]:
191         """
192             Process peripheral chunks according to the configuration.
193
194         Args:
195             chunks (list[dict]): List of chunks to process.
196             config (dict): Configuration for processing peripheral chunks.
197             content_key (str): Key for the content in each chunk.
198             order_key (str): Key for the order of each chunk.
199             reverse (bool, optional): Whether to process chunks in reverse
200             order. Defaults to False.
201
202         Returns:
203             list[str]: List of processed chunk strings.
204         """
205         if reverse:
206             chunks = list(reversed(chunks))
207
208         processed_parts = []
209         included_chunks = []
210         total_chunks = len(chunks)
211
212         head_config = config.get("head", {})
213         tail_config = config.get("tail", {})
214
215         head_count = int(head_config.get("count", 0))
216         tail_count = int(tail_config.get("count", 0))
217         in_skip = False
218         skip_char_count = 0
219
220         for i, chunk in enumerate(chunks):
221             if i < head_count:
222                 section = "head"
223             elif i >= total_chunks - tail_count:
224                 section = "tail"
225             elif "middle" in config:
226                 section = "middle"
227             else:
228                 # Show number of characters skipped
229                 skipped_chars = len(chunk[content_key])
230                 if not in_skip:
231                     skip_char_count = skipped_chars
232                     in_skip = True
233                 else:
234                     skip_char_count += skipped_chars
235
236                 continue
237
238             if in_skip:
239                 processed_parts.append(
```

```

240             f" [...] {skip_char_count} characters skipped ...]"
241         )
242         in_skip = False
243         skip_char_count = 0
244
245         section_config = config.get(section, {})
246         section_content_key = section_config.get("content_key",
247 content_key)
248
249         is_summary = section_content_key != content_key
250         summary_suffix = " (Summary)" if is_summary else ""
251
252         chunk_prefix = f"[Chunk {chunk[order_key]}{summary_suffix}]"
253         processed_parts.extend((chunk_prefix, f"
254 {chunk[section_content_key]}"))
255         included_chunks.append(chunk)
256
257     if in_skip:
258         processed_parts.append(f" [...] {skip_char_count} characters
259 skipped ...]")
260
261     if reverse:
262         processed_parts = list(reversed(processed_parts))
263
264     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.	<i>required</i>
current_index	int	Index of the current chunk being processed.	<i>required</i>
peripheral_config	dict	Configuration for peripheral chunks.	<i>required</i>
content_key	str	Key for the content in each chunk.	<i>required</i>
order_key	str	Key for the order of each chunk.	<i>required</i>

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

Returns:

Name	Type	Description
str	str	Rendered chunk with context and headers.

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

```
113     def render_chunk_with_context(
114         self,
115         chunks: list[dict],
116         current_index: int,
117         peripheral_config: dict,
118         content_key: str,
119         order_key: str,
120         main_chunk_start: str,
121         main_chunk_end: str,
122         doc_header_key: str,
123     ) -> str:
124         """
125             Render a chunk with its peripheral context and headers.
126
127         Args:
128             chunks (list[dict]): List of all chunks in the document.
129             current_index (int): Index of the current chunk being processed.
130             peripheral_config (dict): Configuration for peripheral chunks.
131             content_key (str): Key for the content in each chunk.
132             order_key (str): Key for the order of each chunk.
133             main_chunk_start (str): String to mark the start of the main
134             chunk.
135             main_chunk_end (str): String to mark the end of the main chunk.
136             doc_header_key (str): The key for the headers in the current
137             chunk.
138
139         Returns:
140             str: Rendered chunk with context and headers.
141         """
142
143         # If there are no peripheral chunks, return the main chunk
144         if not peripheral_config:
145             return chunks[current_index][content_key]
146
147         combined_parts = ["--- Previous Context ---"]
148
149         combined_parts.extend(
150             self.process_peripheral_chunks(
151                 chunks[:current_index],
152                 peripheral_config.get("previous", {}),
153                 content_key,
154                 order_key,
155             )
156         )
157         combined_parts.append("--- End Previous Context ---\n")
158
159         # Process main chunk
160         main_chunk = chunks[current_index]
161         if headers := self.render_hierarchy_headers(
162             main_chunk, chunks[: current_index + 1], doc_header_key
163         ):
164             combined_parts.append(headers)
165         combined_parts.extend(
166             (
167                 f"{main_chunk_start}",
168                 f"{main_chunk[content_key]}",
169                 f"{main_chunk_end}",
```

```

170         "\n--- Next Context ---",
171     )
172     )
173     combined_parts.extend(
174       self.process_peripheral_chunks(
175         chunks[current_index + 1 :],
176         peripheral_config.get("next", {}),
177         content_key,
178         order_key,
179       )
180     )
181     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.	<i>required</i>
chunks	list[dict]	List of chunks up to and including the current chunk.	<i>required</i>
doc_header_key	str	The key for the headers in the current chunk.	<i>required</i>

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

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

```
262     def render_hierarchy_headers(
263         self,
264         current_chunk: dict,
265         chunks: list[dict],
266         doc_header_key: str,
267     ) -> str:
268         """
269             Render headers for the current chunk's hierarchy.
270
271         Args:
272             current_chunk (dict): The current chunk being processed.
273             chunks (list[dict]): List of chunks up to and including the
274             current chunk.
275             doc_header_key (str): The key for the headers in the current
276             chunk.
277         Returns:
278             str: Rendered headers in the current chunk's hierarchy.
279         """
280         current_hierarchy = {}
281
282         if doc_header_key is None:
283             return ""
284
285         # Find the largest/highest level in the current chunk
286         current_chunk_headers = current_chunk.get(doc_header_key, [])
287
288         # If there are no headers in the current chunk, return an empty
289         string
290         if not current_chunk_headers:
291             return ""
292
293         highest_level = float("inf") # Initialize with positive infinity
294         for header_info in current_chunk_headers:
295             try:
296                 level = header_info.get("level")
297                 if level is not None and level < highest_level:
298                     highest_level = level
299             except Exception as e:
300                 self.runner.console.log(f"[red]Error processing header: {e}[/red]")
301
302                 self.runner.console.log(f"[red]Header: {header_info}[/red]")
303             return ""
304
305         # If no headers found in the current chunk, set highest_level to None
306         if highest_level == float("inf"):
307             highest_level = None
308
309         for chunk in chunks:
310             for header_info in chunk.get(doc_header_key, []):
311                 try:
312                     header = header_info["header"]
313                     level = header_info["level"]
314                     if header and level:
315                         current_hierarchy[level] = header
316                         # Clear lower levels when a higher level header is found
317                         for lower_level in range(level + 1,
318                             len(current_hierarchy) + 1):
```

```

319             if lower_level in current_hierarchy:
320                 current_hierarchy[lower_level] = None
321         except Exception as e:
322             self.runner.console.log(f"[red]Error processing header:
323 {e}[/red]")
324             self.runner.console.log(f"[red]Header: {header_info}
325 [/red]")
326             return ""
327
328     rendered_headers = [
329         f"{'#' * level} {header}"
330         for level, header in sorted(current_hierarchy.items())
331         if header is not None and (highest_level is None or level <
332         highest_level)
333     ]
334     rendered_headers = " > ".join(rendered_headers)
335     return f"_Current Section:_ {rendered_headers}" if rendered_headers
336     else ""

```

`syntax_check()`

Perform a syntax check on the operation configuration.

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

```

54 def syntax_check(self) -> None:
55     """Perform a syntax check on the operation configuration."""
56     # Validate the schema using Pydantic
57     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:

```
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`

```
6  class UnnestOperation(BaseOperation):
7      """
8          A class that represents an operation to unnest a list-like or
9          dictionary value in a dictionary into multiple dictionaries.
10
11         This operation takes a list of dictionaries and a specified key, and
12         creates new dictionaries based on the value type:
13             - For list-like values: Creates a new dictionary for each element in
14                 the list, copying all other key-value pairs.
15             - For dictionary values: Expands specified fields from the nested
16                 dictionary into the parent dictionary.
17
18         Inherits from:
19             BaseOperation
20
21         Usage:
22             ```python
23             from docetl.operations import UnnestOperation
24
25             # Unnesting a list
26             config_list = {"unnest_key": "tags"}
27             input_data_list = [
28                 {"id": 1, "tags": ["a", "b", "c"]},
29                 {"id": 2, "tags": ["d", "e"]}
30             ]
31
32             unnest_op_list = UnnestOperation(config_list)
33             result_list, _ = unnest_op_list.execute(input_data_list)
34
35             # Result will be:
36             # [
37             #     {"id": 1, "tags": "a"},
38             #     {"id": 1, "tags": "b"},
39             #     {"id": 1, "tags": "c"},
40             #     {"id": 2, "tags": "d"},
41             #     {"id": 2, "tags": "e"}
42             # ]
43
44             # Unnesting a dictionary
45             config_dict = {"unnest_key": "user", "expand_fields": ["name",
46             "age"]}
47             input_data_dict = [
48                 {"id": 1, "user": {"name": "Alice", "age": 30, "email":
49                 "alice@example.com"}},
50                 {"id": 2, "user": {"name": "Bob", "age": 25, "email":
51                 "bob@example.com"}}
52             ]
53
54             unnest_op_dict = UnnestOperation(config_dict)
55             result_dict, _ = unnest_op_dict.execute(input_data_dict)
56
57             # Result will be:
58             # [
59             #     {"id": 1, "name": "Alice", "age": 30, "user": {"name": "Alice",
60             "age": 30, "email": "alice@example.com"}},
61             #     {"id": 2, "name": "Bob", "age": 25, "user": {"name": "Bob",
62             "age": 25, "email": "bob@example.com"}}
63         ]
```

```

63     # ]
64     ``
65     """
66
67     class schema(BaseOperation.schema):
68         type: str = "unnest"
69         unnest_key: str
70         keep_empty: bool | None = None
71         expand_fields: list[str] | None = None
72         recursive: bool | None = None
73         depth: int | None = None
74
75     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
76         """
77             Executes the unnest operation on the input data.
78
79             Args:
80                 input_data (list[dict]): A list of dictionaries to process.
81
82             Returns:
83                 tuple[list[dict], float]: A tuple containing the processed
84                 list of dictionaries
85                 and a float value (always 0 in this implementation).
86
87             Raises:
88                 KeyError: If the specified unnest_key is not found in an
89                 input dictionary.
90                 TypeError: If the value of the unnest_key is not iterable
91                 (list, tuple, set, or dict).
92                 ValueError: If unnesting a dictionary and 'expand_fields' is
93                 not provided in the config.
94
95             The operation supports unnesting of both list-like values and
96             dictionary values:
97
98                 1. For list-like values (list, tuple, set):
99                     Each element in the list becomes a separate dictionary in the
100                     output.
101
102                 2. For dictionary values:
103                     The operation expands specified fields from the nested
104                     dictionary into the parent dictionary.
105                     The 'expand_fields' config parameter must be provided to
106                     specify which fields to expand.
107
108             Examples:
109             ```python
110                 # Unnesting a list
111                 unnest_op = UnnestOperation({"unnest_key": "colors"})
112                 input_data = [
113                     {"id": 1, "colors": ["red", "blue"]},
114                     {"id": 2, "colors": ["green"]}
115                 ]
116                 result, _ = unnest_op.execute(input_data)
117                 # Result will be:
118                 # [
119                 #     {"id": 1, "colors": "red"},
120                 #     {"id": 1, "colors": "blue"},
121                 #     {"id": 2, "colors": "green"}
122                 # ]

```

```

124
125     # Unnesting a dictionary
126     unnest_op = UnnestOperation({"unnest_key": "details",
127 "expand_fields": ["color", "size"]})
128     input_data = [
129         {"id": 1, "details": {"color": "red", "size": "large",
130 "stock": 5}},
131         {"id": 2, "details": {"color": "blue", "size": "medium",
132 "stock": 3}}
133     ]
134     result, _ = unnest_op.execute(input_data)
135     # Result will be:
136     # [
137     #     {"id": 1, "details": {"color": "red", "size": "large",
138 "stock": 5}, "color": "red", "size": "large"},
139     #     {"id": 2, "details": {"color": "blue", "size": "medium",
140 "stock": 3}, "color": "blue", "size": "medium"}
141     # ]
142     ...
143
144     Note: When unnesting dictionaries, the original nested dictionary
145 is preserved in the output,
146     and the specified fields are expanded into the parent dictionary.
147 """
148
149     unnest_key = self.config["unnest_key"]
150     recursive = self.config.get("recursive", False)
151     depth = self.config.get("depth", None)
152     if not depth:
153         depth = 1 if not recursive else float("inf")
154     results = []
155
156     def unnest_recursive(item, key, level=0):
157         if level == 0 and not isinstance(item[key], (list, tuple,
158 set, dict)):
159             raise TypeError(f"Value of unnest key '{key}' is not
160 iterable")
161
162         if level > 0 and not isinstance(item[key], (list, tuple, set,
163 dict)):
164             return [item]
165
166         if level >= depth:
167             return [item]
168
169         if isinstance(item[key], dict):
170             expand_fields = self.config.get("expand_fields")
171             if expand_fields is None:
172                 expand_fields = item[key].keys()
173             new_item = copy.deepcopy(item)
174             for field in expand_fields:
175                 if field in new_item[key]:
176                     new_item[field] = new_item[key][field]
177                 else:
178                     new_item[field] = None
179             return [new_item]
180         else:
181             nested_results = []
182             for value in item[key]:
183                 new_item = copy.deepcopy(item)
184                 new_item[key] = value

```

```

185             if recursive and isinstance(value, (list, tuple, set,
186                                         dict)):
187                 nested_results.extend(
188                     unnest_recursive(new_item, key, level + 1)
189                 )
190             else:
191                 nested_results.append(new_item)
192     return nested_results
193
194     for item in input_data:
195         if unnest_key not in item:
196             raise KeyError(
197                 f"Unnest key '{unnest_key}' not found in item. Other
198                 keys are {item.keys()}"
199             )
200
201             results.extend(unnest_recursive(item, unnest_key))
202
203             if not item[unnest_key] and self.config.get("keep_empty",
204 False):
205                 expand_fields = self.config.get("expand_fields")
206                 new_item = copy.deepcopy(item)
207                 if isinstance(item[unnest_key], dict):
208                     if expand_fields is None:
209                         expand_fields = item[unnest_key].keys()
210                     for field in expand_fields:
211                         new_item[field] = None
212                 else:
213                     new_item[unnest_key] = None
214                 results.append(new_item)
215
216             # Assert that no keys are missing after the operation
217             if results:
218                 original_keys = set(input_data[0].keys())
219                 assert original_keys.issubset(
220                     set(results[0].keys())
221                 ), "Keys lost during unnest operation"
222
223     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.	<i>required</i>

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`

```
66     def execute(self, input_data: list[dict]) -> tuple[list[dict], float]:
67         """
68             Executes the unnest operation on the input data.
69
70             Args:
71                 input_data (list[dict]): A list of dictionaries to process.
72
73             Returns:
74                 tuple[list[dict], float]: A tuple containing the processed list
75                 of dictionaries
76                     and a float value (always 0 in this implementation).
77
78             Raises:
79                 KeyError: If the specified unnest_key is not found in an input
80                 dictionary.
81                 TypeError: If the value of the unnest_key is not iterable (list,
82                 tuple, set, or dict).
83                 ValueError: If unnesting a dictionary and 'expand_fields' is not
84                 provided in the config.
85
86                 The operation supports unnesting of both list-like values and
87                 dictionary values:
88
89                 1. For list-like values (list, tuple, set):
90                     Each element in the list becomes a separate dictionary in the
91                     output.
92
93                 2. For dictionary values:
94                     The operation expands specified fields from the nested dictionary
95                     into the parent dictionary.
96                     The 'expand_fields' config parameter must be provided to specify
97                     which fields to expand.
98
99                 Examples:
100                 ```python
101                 # Unnesting a list
102                 unnest_op = UnnestOperation({"unnest_key": "colors"})
103                 input_data = [
104                     {"id": 1, "colors": ["red", "blue"]},
105                     {"id": 2, "colors": ["green"]}
106                 ]
107                 result, _ = unnest_op.execute(input_data)
108                 # Result will be:
109                 # [
110                 #     {"id": 1, "colors": "red"},
111                 #     {"id": 1, "colors": "blue"},
112                 #     {"id": 2, "colors": "green"}
113                 # ]
114
115                 # Unnesting a dictionary
116                 unnest_op = UnnestOperation({"unnest_key": "details",
117                     "expand_fields": ["color", "size"]})
118                 input_data = [
119                     {"id": 1, "details": {"color": "red", "size": "large", "stock": 5}},
120                     {"id": 2, "details": {"color": "blue", "size": "medium", "stock": 3}}
121                 ]
```

```

123     ]
124     result, _ = unnest_op.execute(input_data)
125     # Result will be:
126     # [
127     #   {"id": 1, "details": {"color": "red", "size": "large", "stock": 5}, "color": "red", "size": "large"}, ...
128     #   {"id": 2, "details": {"color": "blue", "size": "medium", "stock": 3}, "color": "blue", "size": "medium"}
129     # ]
130     ``
131
132
133
134     Note: When unnesting dictionaries, the original nested dictionary is
135     preserved in the output,
136     and the specified fields are expanded into the parent dictionary.
137     """
138
139     unnest_key = self.config["unnest_key"]
140     recursive = self.config.get("recursive", False)
141     depth = self.config.get("depth", None)
142     if not depth:
143         depth = 1 if not recursive else float("inf")
144     results = []
145
146     def unnest_recursive(item, key, level=0):
147         if level == 0 and not isinstance(item[key], (list, tuple, set, dict)):
148             raise TypeError(f"Value of unnest key '{key}' is not iterable")
149
150         if level > 0 and not isinstance(item[key], (list, tuple, set, dict)):
151             return [item]
152
153         if level >= depth:
154             return [item]
155
156         if isinstance(item[key], dict):
157             expand_fields = self.config.get("expand_fields")
158             if expand_fields is None:
159                 expand_fields = item[key].keys()
160             new_item = copy.deepcopy(item)
161             for field in expand_fields:
162                 if field in new_item[key]:
163                     new_item[field] = new_item[key][field]
164                 else:
165                     new_item[field] = None
166             return [new_item]
167         else:
168             nested_results = []
169             for value in item[key]:
170                 new_item = copy.deepcopy(item)
171                 new_item[key] = value
172                 if recursive and isinstance(value, (list, tuple, set, dict)):
173                     nested_results.extend(
174                         unnest_recursive(new_item, key, level + 1)
175                     )
176                 else:
177                     nested_results.append(new_item)
178             return nested_results
179
180
181
182
183

```

```
184     for item in input_data:
185         if unnest_key not in item:
186             raise KeyError(
187                 f"Unnest key '{unnest_key}' not found in item. Other keys
188                 are {item.keys()}"
189             )
190
191         results.extend(unnest_recursive(item, unnest_key))
192
193     if not item[unnest_key] and self.config.get("keep_empty", False):
194         expand_fields = self.config.get("expand_fields")
195         new_item = copy.deepcopy(item)
196         if isinstance(item[unnest_key], dict):
197             if expand_fields is None:
198                 expand_fields = item[unnest_key].keys()
199             for field in expand_fields:
200                 new_item[field] = None
201         else:
202             new_item[unnest_key] = None
203         results.append(new_item)
204
# Assert that no keys are missing after the operation
205 if results:
206     original_keys = set(input_data[0].keys())
207     assert original_keys.issubset(
208         set(results[0].keys())
209     ), "Keys lost during unnest operation"
210
211 return results, 0
```

Optimizers

`docetl.optimizers.map_optimizer.optimizer.MapOptimizer`

A class for optimizing map operations in data processing pipelines.

This optimizer analyzes the input operation configuration and data, and generates optimized plans for executing the operation. It can create plans for chunking, metadata extraction, gleaning, chain decomposition, and parallel execution.

Attributes:

Name	Type	Description
<code>config</code>	<code>dict[str, Any]</code>	The configuration dictionary for the optimizer.
<code>console</code>	<code>Console</code>	A Rich console object for pretty printing.
<code>llm_client</code>	<code>LLMClient</code>	A client for interacting with a language model.
<code>_run_operation</code>	<code>Callable</code>	A function to execute operations.
<code>max_threads</code>	<code>int</code>	The maximum number of threads to use for parallel execution.
<code>timeout</code>	<code>int</code>	The timeout in seconds for operation execution.

Source code in [docetl/optimizers/map_optimizer/optimizer.py](#)

```
20  class MapOptimizer:
21      """
22          A class for optimizing map operations in data processing pipelines.
23
24          This optimizer analyzes the input operation configuration and data,
25          and generates optimized plans for executing the operation. It can
26          create plans for chunking, metadata extraction, gleaning, chain
27          decomposition, and parallel execution.
28
29          Attributes:
30              config (dict[str, Any]): The configuration dictionary for the
31              optimizer.
32              console (Console): A Rich console object for pretty printing.
33              llm_client (LLMClient): A client for interacting with a language
34              model.
35              _run_operation (Callable): A function to execute operations.
36              max_threads (int): The maximum number of threads to use for
37              parallel execution.
38              timeout (int): The timeout in seconds for operation execution.
39
40      """
41
42      def __init__(
43          self,
44          runner,
45          run_operation: Callable,
46          timeout: int = 10,
47          is_filter: bool = False,
48          depth: int = 1,
49      ):
50          """
51              Initialize the MapOptimizer.
52
53              Args:
54                  runner (Runner): The runner object.
55                  run_operation (Callable): A function to execute operations.
56                  timeout (int, optional): The timeout in seconds for operation
57                  execution. Defaults to 10.
58                  is_filter (bool, optional): If True, the operation is a
59                  filter operation. Defaults to False.
60
61                  self.runner = runner
62                  self.config = runner.config
63                  self.console = runner.console
64                  self.llm_client = runner.optimizer.llm_client
65                  self._run_operation = run_operation
66                  self.max_threads = runner.max_threads
67                  self.timeout = runner.optimizer.timeout
68                  self._num_plans_to_evaluate_in_parallel = 5
69                  self.is_filter = is_filter
70                  self.k_to_pairwise_compare = 6
71
72                  self.plan_generator = PlanGenerator(
73                      runner,
74                      self.llm_client,
75                      self.console,
76                      self.config,
```

```

77         run_operation,
78         self.max_threads,
79         is_filter,
80         depth,
81     )
82     self.evaluator = Evaluator(
83         self.llm_client,
84         self.console,
85         self._run_operation,
86         self.timeout,
87         self._num_plans_to_evaluate_in_parallel,
88         self.is_filter,
89     )
90     self.prompt_generator = PromptGenerator(
91         self.runner,
92         self.llm_client,
93         self.console,
94         self.config,
95         self.max_threads,
96         self.is_filter,
97     )
98
99     def should_optimize(
100         self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
101     ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]]]:
102         """
103             Determine if the given operation configuration should be
104             optimized.
105         """
106         (
107             input_data,
108             output_data,
109             ,
110             ,
111             validator_prompt,
112             assessment,
113             data_exceeds_limit,
114         ) = self._should_optimize_helper(op_config, input_data)
115         if data_exceeds_limit or assessment.get("needs_improvement",
116         True):
117             assessment_str = (
118                 "\n".join(assessment.get("reasons", []))
119                 + "\n\nHere are some improvements that may help:\n"
120                 + "\n".join(assessment.get("improvements", [])))
121             )
122             if data_exceeds_limit:
123                 assessment_str += "\nAlso, the input data exceeds the
124                 token limit."
125             return assessment_str, input_data, output_data
126         else:
127             return "", input_data, output_data
128
129     def _should_optimize_helper(
130         self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
131     ) -> tuple[
132         list[dict[str, Any]],
133         list[dict[str, Any]],
134         int,
135         float,
136         str,
137         dict[str, Any],

```

```

138         bool,
139     ]:
140         """
141             Determine if the given operation configuration should be
142             optimized.
143             Create a custom validator prompt and assess the operation's
144             performance
145             using the validator.
146             """
147             self.console.post_optimizer_status(StageType.SAMPLE_RUN)
148             input_data = copy.deepcopy(input_data)
149             # Add id to each input_data
150             for i in range(len(input_data)):
151                 input_data[i]["_map_opt_id"] = str(uuid.uuid4())
152
153             # Define the token limit (adjust as needed)
154             model_input_context_length = model_cost.get(
155                 op_config.get("model", self.config.get("default_model")), {}
156             ).get("max_input_tokens", 8192)
157
158             # Render the prompt with all sample inputs and count tokens
159             total_tokens = 0
160             exceed_count = 0
161             for sample in input_data:
162                 rendered_prompt =
163                     Template(op_config["prompt"]).render(input=sample)
164                     prompt_tokens = count_tokens(
165                         rendered_prompt,
166                         op_config.get("model", self.config.get("default_model")),
167                     )
168                     total_tokens += prompt_tokens
169
170             if prompt_tokens > model_input_context_length:
171                 exceed_count += 1
172
173             # Calculate average tokens and percentage of samples exceeding
174             limit
175             avg_tokens = total_tokens / len(input_data)
176             exceed_percentage = (exceed_count / len(input_data)) * 100
177
178             data_exceeds_limit = exceed_count > 0
179             if exceed_count > 0:
180                 self.console.log(
181                     f"[yellow]Warning: {exceed_percentage:.2f}% of prompts
182 exceed token limit. "
183                     f"Average token count: {avg_tokens:.2f}. "
184                     f"Truncating input data when generating validators.
185 [/yellow]"
186                 )
187
188             # Execute the original operation on the sample data
189             no_change_start = time.time()
190             output_data = self._run_operation(op_config, input_data,
191             is_build=True)
192             no_change_runtime = time.time() - no_change_start
193
194             # Capture output for the sample run
195             self.runner.optimizer.captured_output.save_optimizer_output(
196                 stage_type=StageType.SAMPLE_RUN,
197                 output={
198                     "operation_config": op_config,

```

```
199         "input_data": input_data,
200         "output_data": output_data,
201     },
202 )
203
204     # Generate custom validator prompt
205     self.console.post_optimizer_status(StageType.SHOULD_OPTIMIZE)
206     validator_prompt =
207     self.prompt_generator._generate_validator_prompt(
208         op_config, input_data, output_data
209     )
210
211     # Log the validator prompt
212     self.console.log("[bold]Validator Prompt:[/bold]")
213     self.console.log(validator_prompt)
214     self.console.log("\n") # Add a newline for better readability
215
216     # Step 2: Use the validator prompt to assess the operation's
217     performance
218     assessment = self.evaluator._assess_operation(
219         op_config, input_data, output_data, validator_prompt
220     )
221
222     # Print out the assessment
223     self.console.log(
224         f"[bold]Assessment for whether we should improve operation
{op_config['name']}:[/bold]"
225     )
226     for key, value in assessment.items():
227         self.console.log(f"[bold cyan]{key}:[/bold cyan] [yellow]
{value}[/yellow]")
228         self.console.log("\n") # Add a newline for better readability
229
230     self.runner.optimizer.captured_output.save_optimizer_output(
231         stage_type=StageType.SHOULD_OPTIMIZE,
232         output={
233             "validator_prompt": validator_prompt,
234             "needs_improvement": assessment.get("needs_improvement",
235             True),
236             "reasons": assessment.get("reasons", []),
237             "improvements": assessment.get("improvements", []),
238         },
239     )
240     self.console.post_optimizer_rationale(
241         assessment.get("needs_improvement", True),
242         "\n".join(assessment.get("reasons", []))
243         + "\n\n"
244         + "\n".join(assessment.get("improvements", [])),
245         validator_prompt,
246     )
247
248
249     return (
250         input_data,
251         output_data,
252         model_input_context_length,
253         no_change_runtime,
254         validator_prompt,
255         assessment,
256         data_exceeds_limit,
257     )
258
259
```

```

260     def optimize(
261         self,
262         op_config: dict[str, Any],
263         input_data: list[dict[str, Any]],
264         plan_types: list[str] | None = ["chunk", "proj_synthesis",
265 "glean"],
266         ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
267         """
268             Optimize the given operation configuration for the input data.
269             Uses a staged evaluation approach:
270             1. For data exceeding limits: Try all plan types at once
271             2. For data within limits:
272                 - First try glean/proj synthesis
273                 - Compare with baseline
274                 - Selectively try chunking plans based on initial results
275         """
276         # Verify that the plan types are valid
277         for plan_type in plan_types:
278             if plan_type not in ["chunk", "proj_synthesis", "glean"]:
279                 raise ValueError(
280                     f"Invalid plan type: {plan_type}. Valid plan types
281 are: chunk, proj_synthesis, glean."
282                 )
283
284         (
285             input_data,
286             output_data,
287             model_input_context_length,
288             no_change_runtime,
289             validator_prompt,
290             assessment,
291             data_exceeds_limit,
292         ) = self._should_optimize_helper(op_config, input_data)
293
294         if not self.config.get("optimizer_config",
295             {}).get("force_decompose", False):
296             if not data_exceeds_limit and not
297                 assessment.get("needs_improvement", True):
298                 self.console.log(
299                     f"[green]No improvement needed for operation
300 {op_config['name']}[/green]"
301                 )
302             return (
303                 [op_config],
304                 output_data,
305                 self.plan_generator.subplan_optimizer_cost,
306             )
307
308         # Select consistent evaluation samples
309         num_evaluations = min(5, len(input_data))
310         evaluation_samples = select_evaluation_samples(input_data,
311         num_evaluations)
312
313         if data_exceeds_limit:
314             # For data exceeding limits, try all plan types at once
315             return self._evaluate_all_plans(
316                 op_config,
317                 input_data,
318                 evaluation_samples,
319                 validator_prompt,
320                 plan_types,

```

```

321             model_input_context_length,
322             data_exceeds_limit=True,
323         )
324
325     # For data within limits, use staged evaluation
326     return self._staged_evaluation(
327         op_config,
328         input_data,
329         evaluation_samples,
330         validator_prompt,
331         plan_types,
332         no_change_runtime,
333         model_input_context_length,
334     )
335
336     def _select_best_plan(
337         self,
338         results: dict[str, tuple[float, float, list[dict[str, Any]]]],
339         op_config: dict[str, Any],
340         evaluation_samples: list[dict[str, Any]],
341         validator_prompt: str,
342         candidate_plans: dict[str, list[dict[str, Any]]],
343     ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], str, dict[str, int]]:
344         """
345             Select the best plan from evaluation results using top-k
346             comparison.
347
348             Returns:
349                 Tuple of (best plan, best output, best plan name, pairwise
350                 rankings)
351             """
352
353         # Sort results by score in descending order
354         sorted_results = sorted(results.items(), key=lambda x: x[1][0],
355         reverse=True)
356
357         # Take the top k plans
358         top_plans = sorted_results[: self.k_to_pairwise_compare]
359
360         # Check if there are no top plans
361         if len(top_plans) == 0:
362             raise ValueError(
363                 "No valid plans were generated. Unable to proceed with
364                 optimization."
365             )
366
367         # Include any additional plans that are tied with the last plan
368         tail_score = (
369             top_plans[-1][1][0]
370             if len(top_plans) == self.k_to_pairwise_compare
371             else float("-inf")
372         )
373         filtered_results = dict(
374             top_plans
375             +
376             [
377                 item
378                 for item in sorted_results[len(top_plans):]
379                 if item[1][0] == tail_score
380             ]
381         )

```

```

382         # Perform pairwise comparisons on filtered plans
383         if len(filtered_results) > 1:
384             pairwise_rankings = self.evaluator._pairwise_compare_plans(
385                 filtered_results, validator_prompt, op_config,
386                 evaluation_samples
387             )
388             best_plan_name = max(pairwise_rankings,
389                 key=pairwise_rankings.get)
390         else:
391             pairwise_rankings = {k: 0 for k in results.keys()}
392             best_plan_name = next(iter(filtered_results))
393
394         # Display results table
395         self.console.log(
396             f"\n[bold]Plan Evaluation Results for {op_config['name']} "
397             f"({op_config['type']}, {len(results)} plans, {len(evaluation_samples)} "
398             f"samples):[/bold]"
399             )
400         table = Table(show_header=True, header_style="bold magenta")
401         table.add_column("Plan", style="dim")
402         table.add_column("Score", justify="right", width=10)
403         table.add_column("Runtime", justify="right", width=10)
404         table.add_column("Pairwise Wins", justify="right", width=10)
405
406         for plan_name, (score, runtime, _) in sorted_results:
407             table.add_row(
408                 plan_name,
409                 f"{score:.2f}",
410                 f"{runtime:.2f}s",
411                 f"{pairwise_rankings.get(plan_name, 0)}",
412             )
413
414         self.console.log(table)
415         self.console.log("\n")
416
417     try:
418         best_plan = candidate_plans[best_plan_name]
419         best_output = results[best_plan_name][2]
420     except KeyError:
421         raise ValueError(
422             f"Best plan name {best_plan_name} not found in candidate "
423             f"plans. Candidate plan names: {candidate_plans.keys()}"
424             )
425
426     self.console.log(
427         f"[green]Current best plan: {best_plan_name} for operation "
428             f"{op_config['name']} "
429             f"(Score: {results[best_plan_name][0]:.2f}, "
430             f"Runtime: {results[best_plan_name][1]:.2f}s)[/green]"
431         )
432
433     return best_plan, best_output, best_plan_name, pairwise_rankings
434
435     def _staged_evaluation(
436         self,
437         op_config: dict[str, Any],
438         input_data: list[dict[str, Any]],
439         evaluation_samples: list[dict[str, Any]],
440         validator_prompt: str,
441         plan_types: list[str],
442         no_change_runtime: float,

```

```

443     model_input_context_length: int,
444 ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
445     """Stage 1: Try gleaning and proj synthesis plans first"""
446     candidate_plans = {"no_change": [op_config]}
447
448     # Generate initial plans (gleaning and proj synthesis)
449     if "glean" in plan_types:
450         self.console.log(
451             "[bold magenta]Generating gleaning plans...[/bold magenta]"
452         )
453         )
454         gleaning_plans =
455         self.plan_generator._generate_gleaning_plans(
456             op_config, validator_prompt
457         )
458         candidate_plans.update(gleaning_plans)
459
460     if "proj_synthesis" in plan_types and not self.is_filter:
461         self.console.log(
462             "[bold magenta]Generating independent projection"
463             "synthesis plans...[/bold magenta]"
464         )
465         parallel_plans =
466         self.plan_generator._generate_parallel_plans(
467             op_config, input_data
468         )
469         candidate_plans.update(parallel_plans)
470
471         self.console.log(
472             "[bold magenta]Generating chain projection synthesis"
473             "plans...[/bold magenta]"
474         )
475         chain_plans = self.plan_generator._generate_chain_plans(
476             op_config, input_data
477         )
478         candidate_plans.update(chain_plans)
479
480     # Evaluate initial plans
481     initial_results = self._evaluate_plans(
482         candidate_plans,
483         op_config,
484         evaluation_samples,
485         validator_prompt,
486         no_change_runtime,
487     )
488
489     # Get best initial plan
490     best_plan, best_output, best_plan_name, pairwise_rankings = (
491         self._select_best_plan(
492             initial_results,
493             op_config,
494             evaluation_samples,
495             validator_prompt,
496             candidate_plans,
497         )
498     )
499     best_is_better_than_baseline = best_plan_name != "no_change"
500
501     # Stage 2: Decide whether/how to try chunking plans
502     if "chunk" in plan_types:
503         if best_is_better_than_baseline:

```

```

504         # Try 2 random chunking plans first
505         self.console.log(
506             "[bold magenta]Trying sample of chunking plans...
507 [/bold magenta]"
508         )
509         chunk_plans =
510         self.plan_generator._generate_chunk_size_plans(
511             op_config, input_data, validator_prompt,
512             model_input_context_length
513         )
514
515         if chunk_plans:
516             # Sample 2 random plans
517             chunk_items = list(chunk_plans.items())
518             sample_plans = dict(
519                 random.sample(chunk_items, min(2,
520                 len(chunk_items)))
521             )
522             sample_results = self._evaluate_plans(
523                 sample_plans, op_config, evaluation_samples,
524                 validator_prompt
525             )
526
527             # Do pairwise comparison between sampled plans and
528             current_best
529             current_best = {best_plan_name:
530                 initial_results[best_plan_name]}
531             current_best.update(sample_results)
532
533             _, _, new_best_name, new_pairwise_rankings =
534             self._select_best_plan(
535                 current_best,
536                 op_config,
537                 evaluation_samples,
538                 validator_prompt,
539                 {**{best_plan_name: best_plan}, **sample_plans},
540             )
541
542             if new_best_name == best_plan_name:
543                 self.console.log(
544                     "[yellow]Sample chunking plans did not
545 improve results. Keeping current best plan.[/yellow]"
546                 )
547             return (
548                 best_plan,
549                 best_output,
550                 self.plan_generator.subplan_optimizer_cost,
551             )
552
553             # If a sampled plan wins, evaluate all chunking plans
554             self.console.log(
555                 "[bold magenta]Generating all chunking plans...
556 [/bold magenta]"
557             )
558             chunk_results = self._evaluate_plans(
559                 chunk_plans, op_config, evaluation_samples,
560                 validator_prompt
561             )
562             initial_results.update(chunk_results)
563             candidate_plans.update(chunk_plans)
564         else:

```

```

565         # Try all chunking plans since no improvement found yet
566         self.console.log(
567             "[bold magenta]Generating chunking plans...[/bold"
568             magenta]")
569     )
570     chunk_plans =
571     self.plan_generator._generate_chunk_size_plans(
572         op_config, input_data, validator_prompt,
573         model_input_context_length
574         )
575     chunk_results = self._evaluate_plans(
576         chunk_plans, op_config, evaluation_samples,
577         validator_prompt
578         )
579     initial_results.update(chunk_results)
580     candidate_plans.update(chunk_plans)
581
582     # Final selection of best plan
583     best_plan, best_output, _, final_pairwise_rankings =
584     self._select_best_plan(
585         initial_results,
586         op_config,
587         evaluation_samples,
588         validator_prompt,
589         candidate_plans,
590         )
591
592     # Capture evaluation results with pairwise rankings
593     ratings = {k: v[0] for k, v in initial_results.items()}
594     runtime = {k: v[1] for k, v in initial_results.items()}
595     sample_outputs = {k: v[2] for k, v in initial_results.items()}
596     self.runner.optimizer.captured_output.save_optimizer_output(
597         stage_type=StageType.EVALUATION_RESULTS,
598         output={
599             "input_data": evaluation_samples,
600             "all_plan_ratings": ratings,
601             "all_plan_runtimes": runtime,
602             "all_plan_sample_outputs": sample_outputs,
603             "all_plan_pairwise_rankings": final_pairwise_rankings,
604             },
605         )
606
607     self.console.post_optimizer_status(StageType.END)
608     return best_plan, best_output,
609     self.plan_generator.subplan_optimizer_cost
610
611     def _evaluate_plans(
612         self,
613         plans: dict[str, list[dict[str, Any]]],
614         op_config: dict[str, Any],
615         evaluation_samples: list[dict[str, Any]],
616         validator_prompt: str,
617         no_change_runtime: float | None = None,
618         ) -> dict[str, tuple[float, float, list[dict[str, Any]]]]:
619         """Helper method to evaluate a set of plans in parallel"""
620         results = {}
621         plans_list = list(plans.items())
622
623         for i in range(0, len(plans_list),
624         self._num_plans_to_evaluate_in_parallel):
625             batch = plans_list[i : i +

```

```

626     self._num_plans_to_evaluate_in_parallel]
627         with ThreadPoolExecutor(
628             max_workers=self._num_plans_to_evaluate_in_parallel
629         ) as executor:
630             futures = {
631                 executor.submit(
632                     self.evaluator._evaluate_plan,
633                     plan_name,
634                     op_config,
635                     plan,
636                     copy.deepcopy(evaluation_samples),
637                     validator_prompt,
638                 ): plan_name
639                 for plan_name, plan in batch
640             }
641             for future in as_completed(futures):
642                 plan_name = futures[future]
643                 try:
644                     score, runtime, output =
645                     future.result(timeout=self.timeout)
646                     results[plan_name] = (score, runtime, output)
647                 except concurrent.futures.TimeoutError:
648                     self.console.log(
649                         f"[yellow]Plan {plan_name} timed out and will
650 be skipped.[/yellow]"
651                     )
652                 except Exception as e:
653                     self.console.log(
654                         f"[red]Error in plan {plan_name}: {str(e)}"
655                     )
656
657             if "no_change" in results and no_change_runtime is not None:
658                 results["no_change"] = (
659                     results["no_change"][0],
660                     no_change_runtime,
661                     results["no_change"][2],
662                 )
663
664
665             return results
666
667     def _evaluate_all_plans(
668         self,
669         op_config: dict[str, Any],
670         input_data: list[dict[str, Any]],
671         evaluation_samples: list[dict[str, Any]],
672         validator_prompt: str,
673         plan_types: list[str],
674         model_input_context_length: int,
675         data_exceeds_limit: bool,
676     ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
677         """
678             Evaluate all plans for a given operation configuration.
679         """
680         candidate_plans = {}
681
682         # Generate all plans
683         self.console.post_optimizer_status(StageType.CANDIDATE_PLANS)
684         self.console.log(
685             f"[bold magenta]Generating {len(plan_types)} plans...[/bold
686             magenta]"

```

```

687         )
688     for plan_type in plan_types:
689         if plan_type == "chunk":
690             self.console.log(
691                 "[bold magenta]Generating chunking plans...[/bold"
692                 "magenta]"
693             )
694             chunk_size_plans =
self.plan_generator._generate_chunk_size_plans(
                op_config, input_data, validator_prompt,
model_input_context_length
            )
            candidate_plans.update(chunk_size_plans)
elif plan_type == "proj_synthesis":
    if not self.is_filter:
        self.console.log(
            "[bold magenta]Generating independent projection"
            "synthesis plans...[/bold magenta]"
        )
        parallel_plans =
self.plan_generator._generate_parallel_plans(
                op_config, input_data
            )
        candidate_plans.update(parallel_plans)

        self.console.log(
            "[bold magenta]Generating chain projection"
            "synthesis plans...[/bold magenta]"
        )
        chain_plans =
self.plan_generator._generate_chain_plans(
                op_config, input_data
            )
        candidate_plans.update(chain_plans)
elif plan_type == "glean":
    self.console.log(
        "[bold magenta]Generating gleaning plans...[/bold"
        "magenta]"
    )
    gleaning_plans =
self.plan_generator._generate_gleaning_plans(
                op_config, validator_prompt
            )
    candidate_plans.update(gleaning_plans)

# Capture candidate plans
self.runner.optimizer.captured_output.save_optimizer_output(
    stage_type=StageType.CANDIDATE_PLANS,
    output=candidate_plans,
)

self.console.post_optimizer_status(StageType.EVALUATION_RESULTS)
self.console.log(
    f"[bold magenta]Evaluating {len(candidate_plans)} plans..."
    "[/bold magenta]"
)

results = self._evaluate_plans(
    candidate_plans, op_config, evaluation_samples,
validator_prompt
)

```

```

        # Select best plan using the centralized method
        best_plan, best_output, _, pairwise_rankings =
self._select_best_plan(
            results, op_config, evaluation_samples, validator_prompt,
candidate_plans
        )

        # Capture evaluation results with pairwise rankings
        ratings = {k: v[0] for k, v in results.items()}
        runtime = {k: v[1] for k, v in results.items()}
        sample_outputs = {k: v[2] for k, v in results.items()}
        self.runner.optimizer.captured_output.save_optimizer_output(
            stage_type=StageType.EVALUATION_RESULTS,
            output={
                "input_data": evaluation_samples,
                "all_plan_ratings": ratings,
                "all_plan_runtimes": runtime,
                "all_plan_sample_outputs": sample_outputs,
                "all_plan_pairwise_rankings": pairwise_rankings,
            },
        )

        self.console.post_optimizer_status(StageType.END)
        return best_plan, best_output,
        self.plan_generator.subplan_optimizer_cost
    )
}

```

`__init__(runner, run_operation, timeout=10, is_filter=False, depth=1)`

Initialize the MapOptimizer.

Parameters:

Name	Type	Description	Default
runner	Runner	The runner object.	required
run_operation	Callable	A function to execute operations.	required
timeout	int	The timeout in seconds for operation execution. Defaults to 10.	10
is_filter	bool	If True, the operation is a filter operation. Defaults to False.	False

Source code in `docetl/optimizers/map_optimizer/optimizer.py`

```
39     def __init__(  
40         self,  
41         runner,  
42         run_operation: Callable,  
43         timeout: int = 10,  
44         is_filter: bool = False,  
45         depth: int = 1,  
46     ):  
47         """  
48             Initialize the MapOptimizer.  
49  
50             Args:  
51                 runner (Runner): The runner object.  
52                 run_operation (Callable): A function to execute operations.  
53                 timeout (int, optional): The timeout in seconds for operation  
54                 execution. Defaults to 10.  
55                 is_filter (bool, optional): If True, the operation is a filter  
56                 operation. Defaults to False.  
57             """  
58         self.runner = runner  
59         self.config = runner.config  
60         self.console = runner.console  
61         self.llm_client = runner.optimizer.llm_client  
62         self._run_operation = run_operation  
63         self.max_threads = runner.max_threads  
64         self.timeout = runner.optimizer.timeout  
65         self._num_plans_to_evaluate_in_parallel = 5  
66         self.is_filter = is_filter  
67         self.k_to_pairwise_compare = 6  
68  
69         self.plan_generator = PlanGenerator(  
70             runner,  
71             self.llm_client,  
72             self.console,  
73             self.config,  
74             run_operation,  
75             self.max_threads,  
76             is_filter,  
77             depth,  
78         )  
79         self.evaluator = Evaluator(  
80             self.llm_client,  
81             self.console,  
82             self._run_operation,  
83             self.timeout,  
84             self._num_plans_to_evaluate_in_parallel,  
85             self.is_filter,  
86         )  
87         self.prompt_generator = PromptGenerator(  
88             self.runner,  
89             self.llm_client,  
90             self.console,  
91             self.config,  
92             self.max_threads,  
93             self.is_filter,  
94         )
```

```
optimize(op_config, input_data, plan_types=['chunk', 'proj_synthesis', 'glean'])
```

Optimize the given operation configuration for the input data. Uses a staged evaluation approach:

- 1. For data exceeding limits: Try all plan types at once
- 2. For data within limits:
 - First try glean/proj synthesis - Compare with baseline - Selectively try chunking plans based on initial results

Source code in `docetl/optimizers/map_optimizer/optimizer.py`

```

240     def optimize(
241         self,
242         op_config: dict[str, Any],
243         input_data: list[dict[str, Any]],
244         plan_types: list[str] | None = ["chunk", "proj_synthesis", "glean"],
245     ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
246         """
247             Optimize the given operation configuration for the input data.
248             Uses a staged evaluation approach:
249             1. For data exceeding limits: Try all plan types at once
250             2. For data within limits:
251                 - First try glean/gproj synthesis
252                 - Compare with baseline
253                 - Selectively try chunking plans based on initial results
254         """
255         # Verify that the plan types are valid
256         for plan_type in plan_types:
257             if plan_type not in ["chunk", "proj_synthesis", "glean"]:
258                 raise ValueError(
259                     f"Invalid plan type: {plan_type}. Valid plan types are:
260                     chunk, proj_synthesis, glean."
261                 )
262
263         (
264             input_data,
265             output_data,
266             model_input_context_length,
267             no_change_runtime,
268             validator_prompt,
269             assessment,
270             data_exceeds_limit,
271         ) = self._should_optimize_helper(op_config, input_data)
272
273         if not self.config.get("optimizer_config", {}).get("force_decompose",
274             False):
275             if not data_exceeds_limit and not
276                 assessment.get("needs_improvement", True):
277                 self.console.log(
278                     f"[green]No improvement needed for operation
279                     {op_config['name']}[/green]"
280                 )
281             return (
282                 [op_config],
283                 output_data,
284                 self.plan_generator.subplan_optimizer_cost,
285             )
286
287         # Select consistent evaluation samples
288         num_evaluations = min(5, len(input_data))
289         evaluation_samples = select_evaluation_samples(input_data,
290             num_evaluations)
291
292         if data_exceeds_limit:
293             # For data exceeding limits, try all plan types at once
294             return self._evaluate_all_plans(
295                 op_config,
296                 input_data,

```

```

297         evaluation_samples,
298         validator_prompt,
299         plan_types,
300         model_input_context_length,
301         data_exceeds_limit=True,
302     )
303
304     # For data within limits, use staged evaluation
305     return self._staged_evaluation(
306         op_config,
307         input_data,
308         evaluation_samples,
309         validator_prompt,
310         plan_types,
311         no_change_runtime,
312         model_input_context_length,
313     )

```

`should_optimize(op_config, input_data)`

Determine if the given operation configuration should be optimized.

Source code in `docetl/optimizers/map_optimizer/optimizer.py`

```

94 def should_optimize(
95     self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
96 ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]]]:
97     """
98     Determine if the given operation configuration should be optimized.
99     """
100    (
101        input_data,
102        output_data,
103        -,
104        -,
105        validator_prompt,
106        assessment,
107        data_exceeds_limit,
108    ) = self._should_optimize_helper(op_config, input_data)
109    if data_exceeds_limit or assessment.get("needs_improvement", True):
110        assessment_str = (
111            "\n".join(assessment.get("reasons", []))
112            + "\n\nHere are some improvements that may help:\n"
113            + "\n".join(assessment.get("improvements", []))
114        )
115        if data_exceeds_limit:
116            assessment_str += "\nAlso, the input data exceeds the token
117 limit."
118        return assessment_str, input_data, output_data
119    else:
120        return "", input_data, output_data

```

`docetl.optimizers.reduce_optimizer.ReduceOptimizer`

A class that optimizes reduce operations in data processing pipelines.

This optimizer analyzes the input and output of a reduce operation, creates and evaluates multiple reduce plans, and selects the best plan for optimizing the operation's performance.

Attributes:

Name	Type	Description
config	dict[str, Any]	Configuration dictionary for the optimizer.
console	Console	Rich console object for pretty printing.
llm_client	LLMClient	Client for interacting with a language model.
_run_operation	Callable	Function to run an operation.
max_threads	int	Maximum number of threads to use for parallel processing.
num_fold_prompts	int	Number of fold prompts to generate.
num_samples_in_validation	int	Number of samples to use in validation.

Source code in [docetl/optimizers/reduce_optimizer.py](#)

```
19  class ReduceOptimizer:
20      """
21          A class that optimizes reduce operations in data processing
22          pipelines.
23
24          This optimizer analyzes the input and output of a reduce operation,
25          creates and evaluates
26          multiple reduce plans, and selects the best plan for optimizing the
27          operation's performance.
28
29          Attributes:
30              config (dict[str, Any]): Configuration dictionary for the
31              optimizer.
32                  console (Console): Rich console object for pretty printing.
33                  llm_client (LLMClient): Client for interacting with a language
34              model.
35                  _run_operation (Callable): Function to run an operation.
36                  max_threads (int): Maximum number of threads to use for parallel
37              processing.
38                  num_fold_prompts (int): Number of fold prompts to generate.
39                  num_samples_in_validation (int): Number of samples to use in
40              validation.
41          """
42
43          def __init__(
44              self,
45              runner,
46              run_operation: Callable,
47              num_fold_prompts: int = 1,
48              num_samples_in_validation: int = 10,
49          ):
50              """
51                  Initialize the ReduceOptimizer.
52
53                  Args:
54                      config (dict[str, Any]): Configuration dictionary for the
55                      optimizer.
56                      console (Console): Rich console object for pretty printing.
57                      llm_client (LLMClient): Client for interacting with a
58                      language model.
59                      max_threads (int): Maximum number of threads to use for
60                  parallel processing.
61                      run_operation (Callable): Function to run an operation.
62                      num_fold_prompts (int, optional): Number of fold prompts to
63                  generate. Defaults to 1.
64                      num_samples_in_validation (int, optional): Number of samples
65                  to use in validation. Defaults to 10.
66          """
67          self.runner = runner
68          self.config = self.runner.config
69          self.console = self.runner.console
70          self.llm_client = self.runner.optimizer.llm_client
71          self._run_operation = run_operation
72          self.max_threads = self.runner.max_threads
73          self.num_fold_prompts = num_fold_prompts
74          self.num_samples_in_validation = num_samples_in_validation
75          self.status = self.runner.status
```

```

76
77     def should_optimize_helper(
78         self, op_config: dict[str, Any], input_data: list[dict[str,
79 Any]]]
80         ) -> str:
81             # Check if we're running out of token limits for the reduce
82             prompt
83             model = op_config.get("model", self.config.get("default_model",
84 "gpt-4o-mini"))
85             model_input_context_length = model_cost.get(model, {}).get(
86                 "max_input_tokens", 4096
87             )
88
89             # Find the key with the longest value
90             if op_config["reduce_key"] == ["_all"]:
91                 sample_key = tuple(["_all"])
92             else:
93                 longest_key = max(
94                     op_config["reduce_key"], key=lambda k:
95 len(str(input_data[0][k]))
96                 )
97                 sample_key = tuple(
98                     input_data[0][k] if k == longest_key else input_data[0]
99 [k]
100                 for k in op_config["reduce_key"]
101             )
102
103             # Render the prompt with a sample input
104             prompt_template = Template(op_config["prompt"])
105             sample_prompt = prompt_template.render(
106                 reduce_key=dict(zip(op_config["reduce_key"], sample_key)),
107                 inputs=[input_data[0]],
108             )
109
110             # Count tokens in the sample prompt
111             prompt_tokens = count_tokens(sample_prompt, model)
112
113             self.console.post_optimizer_status(StageType.SAMPLE_RUN)
114             original_output = self._run_operation(op_config, input_data)
115
116             # Step 1: Synthesize a validator prompt
117             self.console.post_optimizer_status(StageType.SHOULD_OPTIMIZE)
118             validator_prompt = self._generate_validator_prompt(
119                 op_config, input_data, original_output
120             )
121
122             # Log the validator prompt
123             self.console.log("[bold]Validator Prompt:[/bold]")
124             self.console.log(validator_prompt)
125             self.console.log("\n")  # Add a newline for better readability
126
127             # Step 2: validate the output
128             validator_inputs = self._create_validation_inputs(
129                 input_data, op_config["reduce_key"]
130             )
131             validation_results = self._validate_reduce_output(
132                 op_config, validator_inputs, original_output,
133                 validator_prompt
134             )
135
136             return (

```

```

137         validation_results,
138         prompt_tokens,
139         model_input_context_length,
140         model,
141         validator_prompt,
142         original_output,
143     )
144
145     def should_optimize(
146         self, op_config: dict[str, Any], input_data: list[dict[str,
147         Any]]]
148         ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]]]:
149         (
150             validation_results,
151             prompt_tokens,
152             model_input_context_length,
153             model,
154             validator_prompt,
155             original_output,
156         ) = self.should_optimize_helper(op_config, input_data)
157         if prompt_tokens * 1.5 > model_input_context_length:
158             return (
159                 "The reduce prompt is likely to exceed the token limit
160             for model {model}." ,
161                 input_data,
162                 original_output,
163             )
164
165             if validation_results.get("needs_improvement", False):
166                 return (
167                     "\n".join(
168                         [
169                             f"Issues: {result['issues']} Suggestions:
170                             {result['suggestions']}"
171                             for result in
172                             validation_results["validation_results"]
173                         ]
174                     ),
175                     input_data,
176                     original_output,
177                 )
178             else:
179                 return "", input_data, original_output
180
181     def optimize(
182         self,
183         op_config: dict[str, Any],
184         input_data: list[dict[str, Any]],
185         level: int = 1,
186     ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
187         """
188             Optimize the reduce operation based on the given configuration
189             and input data.
190
191             This method performs the following steps:
192             1. Run the original operation
193             2. Generate a validator prompt
194             3. Validate the output
195             4. If improvement is needed:
196                 a. Evaluate if decomposition is beneficial
197                 b. If decomposition is beneficial, recursively optimize each

```

```

198     sub-operation
199         c. If not, proceed with single operation optimization
200     5. Run the optimized operation(s)
201
202     Args:
203         op_config (dict[str, Any]): Configuration for the reduce
204         operation.
205         input_data (list[dict[str, Any]]): Input data for the reduce
206         operation.
207
208     Returns:
209         tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A
210         tuple containing the list of optimized configurations
211         and the list of outputs from the optimized operation(s), and
212         the cost of the operation due to synthesizing any resolve operations.
213         """
214         (
215             validation_results,
216             prompt_tokens,
217             model_input_context_length,
218             model,
219             validator_prompt,
220             original_output,
221         ) = self._should_optimize_helper(op_config, input_data)
222
223         # add_map_op = False
224         if prompt_tokens * 2 > model_input_context_length:
225             # add_map_op = True
226             self.console.log(
227                 f"[yellow]Warning: The reduce prompt exceeds the token
228                 limit for model {model}. "
229                 f"Token count: {prompt_tokens}, Limit:
230                 {model_input_context_length}. "
231                 f"Add a map operation to the pipeline.[/yellow]"
232             )
233
234         # # Also query an agent to look at a sample of the inputs and
235         # see if they think a map operation would be helpful
236         # preprocessing_steps = ""
237         # should_use_map, preprocessing_steps = self._should_use_map(
238         #     op_config, input_data
239         # )
240         # if should_use_map or add_map_op:
241         #     # Synthesize a map operation
242         #     # map_prompt, map_output_schema =
243         self._synthesize_map_operation(
244             #         op_config, preprocessing_steps, input_data
245             #     )
246             #     # Change the reduce operation prompt to use the map schema
247             #     new_reduce_prompt =
248         self._change_reduce_prompt_to_use_map_schema(
249             #         op_config["prompt"], map_output_schema
250             #     )
251             #     op_config["prompt"] = new_reduce_prompt
252
253         #     # Return unoptimized map and reduce operations
254         #     return [map_prompt, op_config], input_data, 0.0
255
256         # Print the validation results
257         self.console.log("[bold]Validation Results on Initial Sample:
258         [/bold]")

```

```

259         if validation_results["needs_improvement"] or self.config.get(
260             "optimizer_config", {}
261         ).get("force_decompose", False):
262             self.console.post_optimizer_rationale(
263                 should_optimize=True,
264                 rationale="\n".join(
265                     [
266                         f"Issues: {result['issues']} Suggestions: "
267                         f"{result['suggestions']}"
268                         for result in
269                         validation_results["validation_results"]
270                     ]
271                 ),
272                 validator_prompt=validator_prompt,
273             )
274             self.console.log(
275                 "\n".join(
276                     [
277                         f"Issues: {result['issues']} Suggestions: "
278                         f"{result['suggestions']}"
279                         for result in
280                         validation_results["validation_results"]
281                     ]
282                 )
283             )
284
285         # Step 3: Evaluate if decomposition is beneficial
286         decomposition_result = self._evaluate_decomposition(
287             op_config, input_data, level
288         )
289
290         if decomposition_result["should_decompose"]:
291             return self._optimize_decomposed_reduce(
292                 decomposition_result, op_config, input_data, level
293             )
294
295             return self._optimize_single_reduce(op_config, input_data,
296             validator_prompt)
297         else:
298             self.console.log(f"No improvements identified; "
299             f"{validation_results}.")
299             self.console.post_optimizer_rationale(
300                 should_optimize=False,
301                 rationale="No improvements identified; no optimization "
302                 recommended.,
303                 validator_prompt=validator_prompt,
304             )
305             return [op_config], original_output, 0.0
306
307     def _should_use_map(
308         self, op_config: dict[str, Any], input_data: list[dict[str,
309         Any]]
310     ) -> tuple[bool, str]:
311         """
312             Determine if a map operation should be used based on the input
313             data.
314         """
315
316         # Sample a random input item
317         sample_input = random.choice(input_data)
318
319         # Format the prompt with the sample input

```

```

320     prompt_template = Template(op_config["prompt"])
321     formatted_prompt = prompt_template.render(
322         reduce_key=dict(
323             zip(op_config["reduce_key"],
324                 sample_input[op_config["reduce_key"]])
325         ),
326         inputs=[sample_input],
327     )
328
329     # Prepare the message for the LLM
330     messages = [{"role": "user", "content": formatted_prompt}]
331
332     # Truncate the messages to fit the model's context window
333     truncated_messages = truncate_messages(
334         messages, self.config.get("model", self.default_model)
335     )
336
337     # Query the LLM for preprocessing suggestions
338     preprocessing_prompt = (
339         "Based on the following reduce operation prompt, should we"
340         "do any preprocessing on the input data? "
341         "Consider if we need to remove unnecessary context, or"
342         "logically construct an output that will help in the task. "
343         "If preprocessing would be beneficial, explain why and"
344         "suggest specific steps. If not, explain why preprocessing isn't"
345         "necessary.\n\n"
346         f"Reduce operation prompt:\n{truncated_messages[0]}"
347         '['content']")
348     )
349
350     preprocessing_response = self.llm_client.generate_rewrite(
351         model=self.config.get("model", self.default_model),
352         messages=[{"role": "user", "content":
353             preprocessing_prompt}],
354         response_format={
355             "type": "json_object",
356             "schema": {
357                 "type": "object",
358                 "properties": {
359                     "preprocessing_needed": {"type": "boolean"},
360                     "rationale": {"type": "string"},
361                     "suggested_steps": {"type": "string"},
362                 },
363                 "required": [
364                     "preprocessing_needed",
365                     "rationale",
366                     "suggested_steps",
367                 ],
368                 },
369             },
370         )
371
372     preprocessing_result =
373     preprocessing_response.choices[0].message.content
374
375     should_preprocess = preprocessing_result["preprocessing_needed"]
376     preprocessing_rationale = preprocessing_result["rationale"]
377
378     self.console.log("[bold]Map-Reduce Decomposition Analysis:")
379     self.console.log(f"Should write a map operation:
380

```

```

381     {should_preprocess}")
382         self.console.log(f"Rationale: {preprocessing_rationale}")
383
384         if should_preprocess:
385             self.console.log(
386                 f"Suggested steps:
387 {preprocessing_result['suggested_steps']}"
388             )
389
390         return should_preprocess,
391 preprocessing_result["suggested_steps"]
392
393     def _optimize_single_reduce(
394         self,
395         op_config: dict[str, Any],
396         input_data: list[dict[str, Any]],
397         validator_prompt: str,
398     ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
399         """
400             Optimize a single reduce operation.
401
402             This method performs the following steps:
403             1. Determine and configure value sampling
404             2. Determine if the reduce operation is associative
405             3. Create and evaluate multiple reduce plans
406             4. Run the best reduce plan
407
408             Args:
409                 op_config (dict[str, Any]): Configuration for the reduce
410             operation.
411                 input_data (list[dict[str, Any]]): Input data for the reduce
412             operation.
413                 validator_prompt (str): The validator prompt for evaluating
414             reduce plans.
415
416             Returns:
417                 tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A
418             tuple containing a single-item list with the optimized configuration
419                 and a single-item list with the output from the optimized
420             operation, and the cost of the operation due to synthesizing any resolve
421             operations.
422             """
423
424             # Step 1: Determine and configure value sampling (TODO: re-
425             enable this when the agent is more reliable)
426             # value_sampling_config =
427             self._determine_value_sampling(op_config, input_data)
428             # if value_sampling_config["enabled"]:
429             #     op_config["value_sampling"] = value_sampling_config
430             #     self.console.log("[bold]Value Sampling Configuration:
431             # [/bold]")
432             #     self.console.log(json.dumps(value_sampling_config,
433             indent=2))
434
435             # Step 2: Determine if the reduce operation is associative
436             is_associative = self._is_associative(op_config, input_data)
437
438             # Step 3: Create and evaluate multiple reduce plans
439             self.console.post_optimizer_status(StageType.CANDIDATE_PLANS)
440             self.console.log("[bold magenta]Generating batched plans...
441             [/bold magenta]")
442             reduce_plans = self._create_reduce_plans(op_config, input_data,

```

```

442     is_associative)
443
444         # Create gleaning plans
445         self.console.log("[bold magenta]Generating gleaning plans...[/bold magenta]")
446         gleaning_plans = self._generate_gleaning_plans(reduce_plans,
447                                               validator_prompt)
448
449         self.console.log("[bold magenta]Evaluating plans...[/bold magenta]")
450         self.console.post_optimizer_status(StageType.EVALUATION_RESULTS)
451         best_plan = self._evaluate_reduce_plans(
452             op_config, reduce_plans + gleaning_plans, input_data,
453             validator_prompt
454         )
455
456         # Step 4: Run the best reduce plan
457         optimized_output = self._run_operation(best_plan, input_data)
458         self.console.post_optimizer_status(StageType.END)
459
460         return [best_plan], optimized_output, 0.0
461
462
463     def _generate_gleaning_plans(
464         self,
465         plans: list[dict[str, Any]],
466         validation_prompt: str,
467     ) -> list[dict[str, Any]]:
468         """
469             Generate plans that use gleaning for the given operation.
470
471             Gleaning involves iteratively refining the output of an
472             operation
473             based on validation feedback. This method creates plans with
474             different
475             numbers of gleaning rounds.
476
477             Args:
478                 plans (list[dict[str, Any]]): The list of plans to use for
479                 gleaned.
480                 validation_prompt (str): The prompt used for validating the
481                 operation's output.
482
483             Returns:
484                 dict[str, list[dict[str, Any]]]: A dictionary of gleaned
485                 plans, where each key
486                     is a plan name and each value is a list containing a single
487                     operation configuration
488                     with gleaned parameters.
489
490             """
491
492             # Generate an op with gleaned num_rounds and validation_prompt
493             gleaning_plans = []
494             gleaning_rounds = [1]
495             biggest_batch_size = max([plan["fold_batch_size"] for plan in
496                                       plans])
497             for plan in plans:
498                 if plan["fold_batch_size"] != biggest_batch_size:
499                     continue
500                 for gleaned_round in gleaning_rounds:
501                     plan_copy = copy.deepcopy(plan)
502                     plan_copy["gleaning"] = {

```

```

503             "num_rounds": gleaning_round,
504             "validation_prompt": validation_prompt,
505         }
506         plan_name =
507     f"gleaning_{gleaning_round}_rounds_{plan['name']}"
508         plan_copy["name"] = plan_name
509         gleaning_plans.append(plan_copy)
510     return gleaning_plans
511
512     def _optimize_decomposed_reduce(
513         self,
514         decomposition_result: dict[str, Any],
515         op_config: dict[str, Any],
516         input_data: list[dict[str, Any]],
517         level: int,
518     ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
519         """
520             Optimize a decomposed reduce operation.
521
522             This method performs the following steps:
523             1. Group the input data by the sub-group key.
524             2. Optimize the first reduce operation.
525             3. Run the optimized first reduce operation on all groups.
526             4. Optimize the second reduce operation using the results of the
527                 first.
528             5. Run the optimized second reduce operation.
529
530             Args:
531                 decomposition_result (dict[str, Any]): The result of the
532                 decomposition evaluation.
533                 op_config (dict[str, Any]): The original reduce operation
534                 configuration.
535                 input_data (list[dict[str, Any]]): The input data for the
536                 reduce operation.
537                 level (int): The current level of decomposition.
538             Returns:
539                 tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A
540                 tuple containing the list of optimized configurations
541                     for both reduce operations and the final output of the
542                     second reduce operation, and the cost of the operation due to
543                     synthesizing any resolve operations.
544             """
545             sub_group_key = decomposition_result["sub_group_key"]
546             first_reduce_prompt =
547             decomposition_result["first_reduce_prompt"]
548             second_reduce_prompt =
549             decomposition_result["second_reduce_prompt"]
550             pipeline = []
551             all_cost = 0.0
552
553             first_reduce_config = op_config.copy()
554             first_reduce_config["prompt"] = first_reduce_prompt
555             if isinstance(op_config["reduce_key"], list):
556                 first_reduce_config["reduce_key"] = [sub_group_key] +
557             op_config[
558                 "reduce_key"
559             ]
560             else:
561                 first_reduce_config["reduce_key"] = [sub_group_key,
562             op_config["reduce_key"]]
563                 first_reduce_config["pass_through"] = True

```

```

564
565     if first_reduce_config.get("synthesize_resolve", True):
566         resolve_config = {
567             "name": f"synthesized_resolve_{uuid.uuid4().hex[:8]}",
568             "type": "resolve",
569             "empty": True,
570             "embedding_model": "text-embedding-3-small",
571             "resolution_model": self.config.get("default_model",
572             "gpt-4o-mini"),
573             "comparison_model": self.config.get("default_model",
574             "gpt-4o-mini"),
575             "_intermediates": {
576                 "map_prompt": op_config.get("_intermediates",
577                 {}).get(
578                     "last_map_prompt"
579                 ),
580                 "reduce_key": first_reduce_config["reduce_key"],
581             },
582         }
583         optimized_resolve_config, resolve_cost = JoinOptimizer(
584             self.runner,
585             self.config,
586             resolve_config,
587             self.console,
588             self.llm_client,
589             self.max_threads,
590         ).optimize_resolve(input_data)
591         all_cost += resolve_cost
592
593     if not optimized_resolve_config.get("empty", False):
594         # Add this to the pipeline
595         pipeline += [optimized_resolve_config]
596
597     # Run the resolver
598     optimized_output = self._run_operation(
599         optimized_resolve_config, input_data
600     )
601     input_data = optimized_output
602
603     first_optimized_configs, first_outputs, first_cost =
604     self.optimize(
605         first_reduce_config, input_data, level + 1
606     )
607     pipeline += first_optimized_configs
608     all_cost += first_cost
609
610     # Optimize second reduce operation
611     second_reduce_config = op_config.copy()
612     second_reduce_config["prompt"] = second_reduce_prompt
613     second_reduce_config["pass_through"] = True
614
615     second_optimized_configs, second_outputs, second_cost =
616     self.optimize(
617         second_reduce_config, first_outputs, level + 1
618     )
619
620     # Combine optimized configs and return with final output
621     pipeline += second_optimized_configs
622     all_cost += second_cost
623
624     return pipeline, second_outputs, all_cost

```

```

625
626     def _evaluate_decomposition(
627         self,
628         op_config: dict[str, Any],
629         input_data: list[dict[str, Any]],
630         level: int = 1,
631     ) -> dict[str, Any]:
632         """
633             Evaluate whether decomposing the reduce operation would be
634             beneficial.
635
636             This method first determines if decomposition would be helpful,
637             and if so,
638                 it then determines the sub-group key and prompts for the
639             decomposed operations.
640
641             Args:
642                 op_config (dict[str, Any]): Configuration for the reduce
643             operation.
644                 input_data (list[dict[str, Any]]): Input data for the reduce
645             operation.
646                 level (int): The current level of decomposition.
647
648             Returns:
649                 dict[str, Any]: A dictionary containing the decomposition
650             decision and details.
651             """
652             should_decompose = self._should_decompose(op_config, input_data,
653             level)
654
655             # Log the decomposition decision
656             if should_decompose["should_decompose"]:
657                 self.console.log(
658                     f"[bold green]Decomposition recommended:[/bold green]"
659                     {should_decompose['explanation']}
660                     )
661             else:
662                 self.console.log(
663                     f"[bold yellow]Decomposition not recommended:[/bold"
664                     yellow] {should_decompose['explanation']}"
665                     )
666
667             # Return early if decomposition is not recommended
668             if not should_decompose["should_decompose"]:
669                 return should_decompose
670
671             # Temporarily stop the status
672             if self.status:
673                 self.status.stop()
674
675             # Ask user if they agree with the decomposition assessment
676             user_agrees = Confirm.ask(
677                 f"Do you agree with the decomposition assessment? "
678                 f"[bold]{'Recommended' if"
679                 should_decompose['should_decompose'] else 'Not recommended'}[/bold]",
680                 console=self.console,
681             )
682
683             # If user disagrees, invert the decomposition decision
684             if not user_agrees:
685                 should_decompose["should_decompose"] = not should_decompose[

```

```

686             "should_decompose"
687         ]
688         should_decompose["explanation"] = (
689             "User disagreed with the initial assessment."
690         )
691
692         # Restart the status
693         if self.status:
694             self.status.start()
695
696         # Return if decomposition is not recommended
697         if not should_decompose["should_decompose"]:
698             return should_decompose
699
700         decomposition_details =
701         self._get_decomposition_details(op_config, input_data)
702         result = {**should_decompose, **decomposition_details}
703         if decomposition_details["sub_group_key"] in
704         op_config["reduce_key"]:
705             result["should_decompose"] = False
706             result[
707                 "explanation"
708             ] += " However, the suggested sub-group key is already part
709             of the current reduce key(s), so decomposition is not recommended."
710             result["sub_group_key"] = ""
711
712         return result
713
714     def _should_decompose(
715         self,
716         op_config: dict[str, Any],
717         input_data: list[dict[str, Any]],
718         level: int = 1,
719     ) -> dict[str, Any]:
720         """
721             Determine if decomposing the reduce operation would be
722             beneficial.
723
724             Args:
725                 op_config (dict[str, Any]): Configuration for the reduce
726                 operation.
727                 input_data (list[dict[str, Any]]): Input data for the reduce
728                 operation.
729                 level (int): The current level of decomposition.
730
731             Returns:
732                 dict[str, Any]: A dictionary containing the decomposition
733                 decision and explanation.
734             """
735
736             # TODO: we have not enabled recursive decomposition yet
737             if level > 1 and not op_config.get("recursively_optimize",
738             False):
739                 return {
740                     "should_decompose": False,
741                     "explanation": "Recursive decomposition is not
742                     enabled.",
743                 }
744
745                 system_prompt = (
746                     "You are an AI assistant tasked with optimizing data
747                     processing pipelines."
748

```

```

747         )
748
749     # Sample a subset of input data for analysis
750     sample_size = min(10, len(input_data))
751     sample_input = random.sample(input_data, sample_size)
752
753     # Get all keys from the input data
754     all_keys = set().union(*(item.keys() for item in sample_input))
755     reduce_key = op_config["reduce_key"]
756     reduce_keys = [reduce_key] if isinstance(reduce_key, str) else
757     reduce_key
758     other_keys = [key for key in all_keys if key not in reduce_keys]
759
760     # See if there's an input schema and constrain the sample_input
761     to that schema
762     input_schema = op_config.get("input", {}).get("schema", {})
763     if input_schema:
764         sample_input = [
765             {key: item[key] for key in input_schema} for item in
766             sample_input
767         ]
768
769     # Create a sample of values for other keys
770     sample_values = {
771         key: list(set(str(item.get(key))[:50] for item in
772             sample_input))[:5]
773         for key in other_keys
774     }
775
776     prompt = f"""Analyze the following reduce operation and
777     determine if it should be decomposed into two reduce operations chained
778     together:
779
780     Reduce Operation Prompt:
781     ```
782         {op_config['prompt']}
783     ```
784
785     Current Reduce Key(s): {reduce_keys}
786     Other Available Keys: ', '.join(other_keys)}
787
788     Sample values for other keys:
789     {json.dumps(sample_values, indent=2)}
790
791     Based on this information, determine if it would be beneficial
792     to decompose this reduce operation into a sub-reduce operation followed
793     by a final reduce operation. Consider ALL of the following:
794
795         1. Is there a natural hierarchy in the data (e.g., country ->
796             state -> city) among the other available keys, with a key at a finer
797             level of granularity than the current reduce key(s)?
798         2. Are the current reduce key(s) some form of ID, and are there
799             many different types of inputs for that ID among the other available
800             keys?
801         3. Does the prompt implicitly ask for sub-grouping based on the
802             other available keys (e.g., "summarize policies by state, then by
803             country")?
804         4. Would splitting the operation improve accuracy (i.e., make
805             sure information isn't lost when reducing)?
806         5. Are all the keys of the potential hierarchy provided in the
807             other available keys? If not, we should not decompose.

```

```

808             6. Importantly, do not suggest decomposition using any key that
809                 is already part of the current reduce key(s). We are looking for a new
810                 key from the other available keys to use for sub-grouping.
811             7. Do not suggest keys that don't contain meaningful information
812                 (e.g., id-related keys).
813
814             Provide your analysis in the following format:
815             """
816
817             parameters = {
818                 "type": "object",
819                 "properties": {
820                     "should_decompose": {"type": "boolean"},
821                     "explanation": {"type": "string"},
822                 },
823                 "required": ["should_decompose", "explanation"],
824             }
825
826             response = self.llm_client.generate_rewrite(
827                 [{"role": "user", "content": prompt}],
828                 system_prompt,
829                 parameters,
830             )
831             return json.loads(response.choices[0].message.content)
832
833     def _get_decomposition_details(
834         self,
835         op_config: dict[str, Any],
836         input_data: list[dict[str, Any]],
837     ) -> dict[str, Any]:
838         """
839             Determine the sub-group key and prompts for decomposed reduce
840             operations.
841
842             Args:
843                 op_config (dict[str, Any]): Configuration for the reduce
844                 operation.
845                 input_data (list[dict[str, Any]]): Input data for the reduce
846                 operation.
847
848             Returns:
849                 dict[str, Any]: A dictionary containing the sub-group key
850                 and prompts for decomposed operations.
851             """
852             system_prompt = (
853                 "You are an AI assistant tasked with optimizing data
854                 processing pipelines."
855             )
856
857             # Sample a subset of input data for analysis
858             sample_size = min(10, len(input_data))
859             sample_input = random.sample(input_data, sample_size)
860
861             # Get all keys from the input data
862             all_keys = set().union(*(item.keys() for item in sample_input))
863             reduce_key = op_config["reduce_key"]
864             reduce_keys = [reduce_key] if isinstance(reduce_key, str) else
865             reduce_key
866             other_keys = [key for key in all_keys if key not in reduce_keys]
867
868             prompt = f"""Given that we've decided to decompose the following

```

```

869     reduce operation, suggest a two-step reduce process:
870
871         Reduce Operation Prompt:
872         ``
873         {op_config['prompt']}
874         ``
875
876         Reduce Key(s): {reduce_key}
877         Other Keys: {', '.join(other_keys)}
878
879         Provide the following:
880         1. A sub-group key to use for the first reduce operation
881         2. A prompt for the first reduce operation
882         3. A prompt for the second (final) reduce operation
883
884         For the reduce operation prompts, you should only minimally
885         modify the original prompt. The prompts should be Jinja templates, and
886         the only variables they can access are the `reduce_key` and `inputs`
887         variables.
888
889         Provide your suggestions in the following format:
890         """
891
892             parameters = {
893                 "type": "object",
894                 "properties": {
895                     "sub_group_key": {"type": "string"},
896                     "first_reduce_prompt": {"type": "string"},
897                     "second_reduce_prompt": {"type": "string"},
898                 },
899                 "required": [
900                     "sub_group_key",
901                     "first_reduce_prompt",
902                     "second_reduce_prompt",
903                 ],
904             }
905
906             response = self.llm_client.generate_rewrite(
907                 [{"role": "user", "content": prompt}],
908                 system_prompt,
909                 parameters,
910             )
911             return json.loads(response.choices[0].message.content)
912
913     def _determine_value_sampling(
914         self, op_config: dict[str, Any], input_data: list[dict[str,
915         Any]]
916     ) -> dict[str, Any]:
917         """
918             Determine whether value sampling should be enabled and configure
919             its parameters.
920             """
921             system_prompt = (
922                 "You are an AI assistant helping to optimize data processing
923                 pipelines."
924             )
925
926             # Sample a subset of input data for analysis
927             sample_size = min(100, len(input_data))
928             sample_input = random.sample(input_data, sample_size)
929

```

```

930     prompt = f"""
931         Analyze the following reduce operation and determine if value
932         sampling should be enabled:
933
934         Reduce Operation Prompt:
935         {op_config['prompt']}
936
937         Sample Input Data (first 2 items):
938         {json.dumps(sample_input[:2], indent=2)}
939
940         Value sampling is appropriate for reduce operations that don't
941         need to look at all the values for each key to produce a good result,
942         such as generic summarization tasks.
943
944         Based on the reduce operation prompt and the sample input data,
945         determine if value sampling should be enabled.
946         Answer with 'yes' if value sampling should be enabled or 'no' if
947         it should not be enabled. Explain your reasoning briefly.
948         """
949
950         parameters = {
951             "type": "object",
952             "properties": {
953                 "enable_sampling": {"type": "boolean"},
954                 "explanation": {"type": "string"},
955             },
956             "required": ["enable_sampling", "explanation"],
957         }
958
959         response = self.llm_client.generate_rewrite(
960             [{"role": "user", "content": prompt}],
961             system_prompt,
962             parameters,
963         )
964         result = json.loads(response.choices[0].message.content)
965
966         if not result["enable_sampling"]:
967             return {"enabled": False}
968
969         # Print the explanation for enabling value sampling
970         self.console.log(f"Value sampling enabled:\n{result['explanation']}")
971
972         # Determine sampling method
973         prompt = f"""
974             We are optimizing a reduce operation in a data processing
975             pipeline. The reduce operation is defined by the following prompt:
976
977             Reduce Operation Prompt:
978             {op_config['prompt']}
979
980             Sample Input Data (first 2 items):
981             {json.dumps(sample_input[:2], indent=2)}
982
983             We have determined that value sampling should be enabled for
984             this reduce operation. Value sampling is a technique used to process
985             only a subset of the input data for each reduce key, rather than
986             processing all items. This can significantly reduce processing time and
987             costs for very large datasets, especially when the reduce operation
988             doesn't require looking at every single item to produce a good result
989             (e.g., summarization tasks).
990

```

```

991
992     Now we need to choose the most appropriate sampling method. The
993     available methods are:
994
995         1. "random": Randomly select a subset of values.
996             Example: In a customer review analysis task, randomly selecting
997             a subset of reviews to summarize the overall sentiment.
998
999         2. "cluster": Use K-means clustering to select representative
1000            samples.
1001             Example: In a document categorization task, clustering documents
1002             based on their content and selecting representative documents from each
1003             cluster to determine the overall categories.
1004
1005         3. "sem_sim": Use semantic similarity to select the most
1006            relevant samples to a query text.
1007             Example: In a news article summarization task, selecting
1008             articles that are semantically similar to a query like "Major economic
1009             events of {{reduce_key}}" to produce a focused summary.
1010
1011             Based on the reduce operation prompt, the nature of the task,
1012             and the sample input data, which sampling method would be most
1013             appropriate?
1014
1015             Provide your answer as either "random", "cluster", or "sem_sim",
1016             and explain your reasoning in detail. Consider the following in your
1017             explanation:
1018                 - The nature of the reduce task (e.g., summarization,
1019                   aggregation, analysis)
1020                 - The structure and content of the input data
1021                 - The potential benefits and drawbacks of each sampling method
1022             for this specific task
1023             """
1024
1025             parameters = {
1026                 "type": "object",
1027                 "properties": {
1028                     "method": {"type": "string", "enum": ["random",
1029 "cluster", "sem_sim"]},
1030                     "explanation": {"type": "string"},
1031                 },
1032                 "required": ["method", "explanation"],
1033             }
1034
1035             response = self.llm_client.generate_rewrite(
1036                 [{"role": "user", "content": prompt}],
1037                 system_prompt,
1038                 parameters,
1039             )
1040             result = json.loads(response.choices[0].message.content)
1041             method = result["method"]
1042
1043             value_sampling_config = {
1044                 "enabled": True,
1045                 "method": method,
1046                 "sample_size": 100, # Default sample size
1047                 "embedding_model": "text-embedding-3-small",
1048             }
1049
1050             if method in ["cluster", "sem_sim"]:
1051                 # Determine embedding keys

```

```

1052             prompt = f"""
1053                 For the {method} sampling method, we need to determine which
1054                 keys from the input data should be used for generating embeddings.
1055
1056                 Input data keys:
1057                 {', '.join(sample_input[0].keys())}
1058
1059                 Sample Input Data:
1060                 {json.dumps(sample_input[0], indent=2)[:1000]}...
1061
1062                     Based on the reduce operation prompt and the sample input
1063                     data, which keys should be used for generating embeddings? Use keys that
1064                     will create meaningful embeddings (i.e., not id-related keys).
1065                     Provide your answer as a list of key names that is a subset
1066                     of the input data keys. You should pick only the 1-3 keys that are
1067                     necessary for generating meaningful embeddings, that have relatively
1068                     short values.
1069                     """
1070
1071             parameters = {
1072                 "type": "object",
1073                 "properties": {
1074                     "embedding_keys": {"type": "array", "items":
1075 {"type": "string"}}, "explanation": {"type": "string"}, },
1076                 "required": ["embedding_keys", "explanation"], }
1077
1078             response = self.llm_client.generate_rewrite(
1079                 [{"role": "user", "content": prompt}], system_prompt,
1080                 parameters, )
1081             result = json.loads(response.choices[0].message.content)
1082             # TODO: validate that these exist
1083             embedding_keys = result["embedding_keys"]
1084             for key in result["embedding_keys"]:
1085                 if key not in sample_input[0]:
1086                     embedding_keys.remove(key)
1087
1088             if not embedding_keys:
1089                 # Select the reduce key
1090                 self.console.log(
1091                     "No embedding keys found, selecting reduce key for
1092                     embedding key")
1093             )
1094             embedding_keys = (
1095                 op_config["reduce_key"]
1096                 if isinstance(op_config["reduce_key"], list)
1097                 else [op_config["reduce_key"]]
1098             )
1099
1100             value_sampling_config["embedding_keys"] = embedding_keys
1101
1102             if method == "sem_sim":
1103                 # Determine query text
1104                 prompt = f"""
1105                     For the semantic similarity (sem_sim) sampling method, we
1106                     need to determine the query text to compare against when selecting
1107                     samples.

```

```

1113
1114     Reduce Operation Prompt:
1115     {op_config['prompt']}
1116
1117     The query text should be a Jinja template with access to the
1118     `reduce_key` variable.
1119     Based on the reduce operation prompt, what would be an
1120     appropriate query text for selecting relevant samples?
1121     """
1122
1123     parameters = {
1124         "type": "object",
1125         "properties": {
1126             "query_text": {"type": "string"},
1127             "explanation": {"type": "string"},
1128         },
1129         "required": ["query_text", "explanation"],
1130     }
1131
1132     response = self.llm_client.generate_rewrite(
1133         [{"role": "user", "content": prompt}],
1134         system_prompt,
1135         parameters,
1136     )
1137     result = json.loads(response.choices[0].message.content)
1138     value_sampling_config["query_text"] = result["query_text"]
1139
1140     return value_sampling_config
1141
1142     def _is_associative(
1143         self, op_config: dict[str, Any], input_data: list[dict[str,
1144         Any]]
1145     ) -> bool:
1146         """
1147             Determine if the reduce operation is associative.
1148
1149             This method analyzes the reduce operation configuration and a
1150             sample of the input data
1151             to determine if the operation is associative (i.e., the order of
1152             combining elements
1153             doesn't affect the final result).
1154
1155             Args:
1156                 op_config (dict[str, Any]): Configuration for the reduce
1157             operation.
1158                 input_data (list[dict[str, Any]]): Input data for the reduce
1159             operation.
1160
1161             Returns:
1162                 bool: True if the operation is determined to be associative,
1163             False otherwise.
1164             """
1165             system_prompt = (
1166                 "You are an AI assistant helping to optimize data processing
1167                 pipelines."
1168             )
1169
1170             # Sample a subset of input data for analysis
1171             sample_size = min(5, len(input_data))
1172             sample_input = random.sample(input_data, sample_size)
1173

```

```

1174     prompt = f"""
1175         Analyze the following reduce operation and determine if it is
1176         associative:
1177
1178             Reduce Operation Prompt:
1179             {op_config['prompt']}
1180
1181             Sample Input Data:
1182             fjson.dumps(sample_input, indent=2)[:1000]}...
1183
1184             Based on the reduce operation prompt, determine whether the
1185             order in which we process data matters.
1186             Answer with 'yes' if order matters or 'no' if order doesn't
1187             matter.
1188             Explain your reasoning briefly.
1189
1190             For example:
1191             - Merging extracted key-value pairs from documents does not
1192             require order: combining >{"name": "John", "age": 30} with >{"city": "New York", "job": "Engineer"} yields the same result regardless of
1193             order
1194                 - Generating a timeline of events requires order: the order of
1195             events matters for maintaining chronological accuracy.
1196
1197             Consider these examples when determining whether the order in
1198             which we process data matters. You might also have to consider the
1199             specific data.
1200             """
1201
1202
1203     parameters = {
1204         "type": "object",
1205         "properties": {
1206             "order_matters": {"type": "boolean"},
1207             "explanation": {"type": "string"},
1208         },
1209         "required": ["order_matters", "explanation"],
1210     }
1211
1212     response = self.llm_client.generate_rewrite(
1213         [{"role": "user", "content": prompt}],
1214         system_prompt,
1215         parameters,
1216     )
1217     result = json.loads(response.choices[0].message.content)
1218     result["is_associative"] = not result["order_matters"]
1219
1220     self.console.log(
1221         f"[yellow]Reduce operation {'is associative' if
result['is_associative'] else 'is not associative'}.[/yellow] Analysis:
1222 {result['explanation']}"
1223         )
1224     return result["is_associative"]
1225
1226
1227     def _generate_validator_prompt(
1228         self,
1229         op_config: dict[str, Any],
1230         input_data: list[dict[str, Any]],
1231         original_output: list[dict[str, Any]],
1232     ) -> str:
1233         """
1234             Generate a custom validator prompt for assessing the quality of

```

```
1235     the reduce operation output.  
1236  
1237         This method creates a prompt that will be used to validate the  
1238         output of the reduce operation.  
1239             It includes specific questions about the quality and  
1240             completeness of the output.  
1241  
1242             Args:  
1243                 op_config (dict[str, Any]): Configuration for the reduce  
1244                 operation.  
1245                 input_data (list[dict[str, Any]]): Input data for the reduce  
1246                 operation.  
1247                 original_output (list[dict[str, Any]]): Original output of  
1248                 the reduce operation.  
1249  
1250             Returns:  
1251                 str: A custom validator prompt as a string.  
1252                 """  
1253                     system_prompt = "You are an AI assistant tasked with creating  
1254                     custom validation prompts for reduce operations in data processing  
1255                     pipelines."  
1256  
1257                     sample_input = random.choice(input_data)  
1258                     input_keys = op_config.get("input", {}).get("schema", {})  
1259                     if input_keys:  
1260                         sample_input = {k: sample_input[k] for k in input_keys}  
1261  
1262                     reduce_key = op_config.get("reduce_key")  
1263                     if reduce_key and original_output:  
1264                         if isinstance(reduce_key, list):  
1265                             key = next(  
1266                             (  
1267                                 tuple(item[k] for k in reduce_key)  
1268                                 for item in original_output  
1269                                 if all(k in item for k in reduce_key))  
1270                             ),  
1271                             tuple(None for _ in reduce_key),  
1272                         )  
1273                         sample_output = next(  
1274                         (  
1275                             item  
1276                             for item in original_output  
1277                             if all(item.get(k) == v for k, v in  
1278                             zip(reduce_key, key))  
1279                             ),  
1280                             {},  
1281                         ))  
1282                     else:  
1283                         key = next(  
1284                         (  
1285                             item[reduce_key]  
1286                             for item in original_output  
1287                             if reduce_key in item  
1288                         ),  
1289                         None,  
1290                     ))  
1291                     sample_output = next(  
1292                         (item for item in original_output if  
1293                         item.get(reduce_key) == key),  
1294                         {},  
1295                     ))
```

```

1296         else:
1297             sample_output = original_output[0] if original_output else
1298     {}
1299
1300     output_keys = op_config.get("output", {}).get("schema", {})
1301     sample_output = {k: sample_output[k] for k in output_keys}
1302
1303     prompt = f"""
1304     Analyze the following reduce operation and its input/output:
1305
1306     Reduce Operation Prompt:
1307     {op_config["prompt"]}
1308
1309     Sample Input (just one item):
1310     {json.dumps(sample_input, indent=2)}
1311
1312     Sample Output:
1313     {json.dumps(sample_output, indent=2)}
1314
1315     Create a custom validator prompt that will assess how well the
1316     reduce operation performed its intended task. The prompt should ask
1317     specific 2-3 questions about the quality of the output, such as:
1318         1. Does the output accurately reflect the aggregation method
1319             specified in the task? For example, if finding anomalies, are the
1320                 identified anomalies actually anomalies?
1321         2. Are there any missing fields, unexpected null values, or data
1322             type mismatches in the output compared to the expected schema?
1323         3. Does the output maintain the key information from the input
1324             while appropriately condensing or summarizing it? For instance, in a
1325                 text summarization task, are the main points preserved?
1326         4. How well does the output adhere to any specific formatting
1327             requirements mentioned in the original prompt, such as character limits
1328             for summaries or specific data types for aggregated values?
1329
1330     Note that the output may reflect more than just the input
1331     provided, since we only provide a one-item sample input. Provide your
1332     response as a single string containing the custom validator prompt. The
1333     prompt should be tailored to the task and avoid generic criteria. The
1334     prompt should not reference a specific value in the sample input, but
1335     rather a general property.
1336
1337     Your prompt should not have any placeholders like {{ reduce_key
1338 }} or {{ input_key }}. It should just be a string.
1339     """
1340
1341     parameters = {
1342         "type": "object",
1343         "properties": {"validator_prompt": {"type": "string"}},
1344         "required": ["validator_prompt"],
1345     }
1346
1347     response = self.llm_client.generate_rewrite(
1348         [{"role": "user", "content": prompt}],
1349         system_prompt,
1350         parameters,
1351     )
1352     return json.loads(response.choices[0].message.content)
1353     ["validator_prompt"]
1354
1355     def _validate_reduce_output(
1356         self,

```

```

1357         op_config: dict[str, Any],
1358         validation_inputs: dict[Any, list[dict[str, Any]]],
1359         output_data: list[dict[str, Any]],
1360         validator_prompt: str,
1361     ) -> dict[str, Any]:
1362     """
1363         Validate the output of the reduce operation using the generated
1364         validator prompt.
1365
1366         This method assesses the quality of the reduce operation output
1367         by applying the validator prompt
1368             to multiple samples of the input and output data.
1369
1370         Args:
1371             op_config (dict[str, Any]): Configuration for the reduce
1372             operation.
1373             validation_inputs (dict[Any, list[dict[str, Any]]]): Validation
1374             inputs for the reduce operation.
1375             output_data (list[dict[str, Any]]): Output data from the
1376             reduce operation.
1377             validator_prompt (str): The validator prompt generated
1378             earlier.
1379
1380         Returns:
1381             dict[str, Any]: A dictionary containing validation results
1382             and a flag indicating if improvement is needed.
1383             """
1384             system_prompt = "You are an AI assistant tasked with validating
1385             the output of reduce operations in data processing pipelines."
1386
1387             validation_results = []
1388             with ThreadPoolExecutor(max_workers=self.max_threads) as
1389             executor:
1390                 futures = []
1391                 for reduce_key, inputs in validation_inputs.items():
1392                     if (
1393                         op_config["reduce_key"] == ["_all"]
1394                         or op_config["reduce_key"] == "_all"
1395                     ):
1396                         sample_output = output_data[0]
1397                     elif isinstance(op_config["reduce_key"], list):
1398                         sample_output = next(
1399                             (
1400                                 item
1401                                 for item in output_data
1402                                 if all(
1403                                     item[key] == reduce_key[i]
1404                                     for i, key in
1405                                     enumerate(op_config["reduce_key"])
1406                                 )
1407                             ),
1408                             None,
1409                         )
1410                     else:
1411                         sample_output = next(
1412                             (
1413                                 item
1414                                 for item in output_data
1415                                 if item[op_config["reduce_key"]] ==
1416                                 reduce_key
1417                             ),

```

```

1418                 None,
1419             )
1420
1421         if sample_output is None:
1422             self.console.log(
1423                 f"Warning: No output found for reduce key
1424 {reduce_key}"
1425             )
1426         continue
1427
1428         input_str = json.dumps(inputs, indent=2)
1429         # truncate input_str to 40,000 words
1430         input_str = input_str.split()[:40000]
1431         input_str = " ".join(input_str) + "..."
1432
1433         prompt = f"""{validator_prompt}
1434
1435             Reduce Operation Task:
1436             {op_config["prompt"]}
1437
1438             Input Data Samples:
1439             {input_str}
1440
1441             Output Data Sample:
1442             {json.dumps(sample_output, indent=2)}
1443
1444             Based on the validator prompt and the input/output
1445             samples, assess the quality (e.g., correctness, completeness) of the
1446             reduce operation output.
1447             Provide your assessment in the following format:
1448             """
1449
1450             parameters = {
1451                 "type": "object",
1452                 "properties": {
1453                     "is_correct": {"type": "boolean"},
1454                     "issues": {"type": "array", "items": {"type":
1455 "string"}},
1456                     "suggestions": {"type": "array", "items": {
1457 "type": "string"}},
1458                     },
1459                     "required": ["is_correct", "issues", "suggestions"]
1460                 }
1461
1462             futures.append(
1463                 executor.submit(
1464                     self.llm_client.generate_judge,
1465                     [{"role": "user", "content": prompt}],
1466                     system_prompt,
1467                     parameters,
1468                     )
1469                 )
1470
1471         for future, (reduce_key, inputs) in zip(futures,
1472 validation_inputs.items()):
1473             response = future.result()
1474             result = json.loads(response.choices[0].message.content)
1475             validation_results.append(result)
1476
1477             # Determine if optimization is needed based on validation
1478             results

```

```

1479         invalid_count = sum(
1480             1 for result in validation_results if not
1481             result["is_correct"]
1482         )
1483         needs_improvement = invalid_count > 1 or (
1484             invalid_count == 1 and len(validation_results) == 1
1485         )
1486
1487     return {
1488         "needs_improvement": needs_improvement,
1489         "validation_results": validation_results,
1490     }
1491
1492     def _create_validation_inputs(
1493         self, input_data: list[dict[str, Any]], reduce_key: str |
1494         list[str]
1495     ) -> dict[Any, list[dict[str, Any]]]:
1496         # Group input data by reduce_key
1497         grouped_data = {}
1498         if reduce_key == ["_all"]:
1499             # Put all data in one group under a single key
1500             grouped_data[("_all",)] = input_data
1501         else:
1502             # Group by reduce key(s) as before
1503             for item in input_data:
1504                 if isinstance(reduce_key, list):
1505                     key = tuple(item[k] for k in reduce_key)
1506                 else:
1507                     key = item[reduce_key]
1508                 if key not in grouped_data:
1509                     grouped_data[key] = []
1510                     grouped_data[key].append(item)
1511
1512         # Select a fixed number of reduce keys
1513         selected_keys = random.sample(
1514             list(grouped_data.keys()),
1515             min(self.num_samples_in_validation, len(grouped_data)),
1516         )
1517
1518         # Create a new dict with only the selected keys
1519         validation_inputs = {key: grouped_data[key] for key in
1520             selected_keys}
1521
1522         return validation_inputs
1523
1524     def _create_reduce_plans(
1525         self,
1526         op_config: dict[str, Any],
1527         input_data: list[dict[str, Any]],
1528         is_associative: bool,
1529     ) -> list[dict[str, Any]]:
1530         """
1531             Create multiple reduce plans based on the input data and
1532             operation configuration.
1533
1534             This method generates various reduce plans by varying batch
1535             sizes and fold prompts.
1536             It takes into account the LLM's context window size to determine
1537             appropriate batch sizes.
1538
1539             Args:

```

```

1540         op_config (dict[str, Any]): Configuration for the reduce
1541     operation.
1542     input_data (list[dict[str, Any]]): Input data for the reduce
1543     operation.
1544     is_associative (bool): Flag indicating whether the reduce
1545     operation is associative.
1546
1547     Returns:
1548     list[dict[str, Any]]: A list of reduce plans, each with
1549     different batch sizes and fold prompts.
1550     """
1551     model = op_config.get("model", "gpt-4o-mini")
1552     model_input_context_length = model_cost.get(model, {}).get(
1553         "max_input_tokens", 8192
1554     )
1555
1556     # Estimate tokens for prompt, input, and output
1557     prompt_tokens = count_tokens(op_config["prompt"], model)
1558     sample_input = input_data[:100]
1559     sample_output = self._run_operation(op_config, input_data[:100])
1560
1561     prompt_vars = extract_jinja_variables(op_config["prompt"])
1562     prompt_vars = [var.split(".")[-1] for var in prompt_vars]
1563     avg_input_tokens = mean(
1564         [
1565             count_tokens(
1566                 json.dumps({k: item[k] for k in prompt_vars if k in
1567                 item}), model
1568             )
1569             for item in sample_input
1570         ]
1571     )
1572     avg_output_tokens = mean(
1573         [
1574             count_tokens(
1575                 json.dumps({k: item[k] for k in prompt_vars if k in
1576                 item}), model
1577             )
1578             for item in sample_output
1579         ]
1580     )
1581
1582     # Calculate max batch size that fits in context window
1583     max_batch_size = (
1584         model_input_context_length - prompt_tokens -
1585         avg_output_tokens
1586         ) // avg_input_tokens
1587
1588     # Generate 6 candidate batch sizes
1589     batch_sizes = [
1590         max(1, int(max_batch_size * ratio))
1591         for ratio in [0.1, 0.2, 0.4, 0.6, 0.75, 0.9]
1592     ]
1593     # Log the generated batch sizes
1594     self.console.log("[cyan]Generating plans for batch sizes:
1595     [/cyan]")
1596     for size in batch_sizes:
1597         self.console.log(f" - {size}")
1598     batch_sizes = sorted(set(batch_sizes)) # Remove duplicates and
1599     sort
1600

```

```

1601     plans = []
1602
1603     # Generate multiple fold prompts
1604     max_retries = 5
1605     retry_count = 0
1606     fold_prompts = []
1607
1608     while retry_count < max_retries and not fold_prompts:
1609         try:
1610             fold_prompts = self._synthesize_fold_prompts(
1611                 op_config,
1612                 sample_input,
1613                 sample_output,
1614                 num_prompts=self.num_fold_prompts,
1615             )
1616             fold_prompts = list(set(fold_prompts))
1617             if not fold_prompts:
1618                 raise ValueError("No fold prompts generated")
1619         except Exception as e:
1620             retry_count += 1
1621             if retry_count == max_retries:
1622                 raise RuntimeError(
1623                     f"Failed to generate fold prompts after "
1624                     f"{max_retries} attempts: {str(e)}"
1625                 )
1626             self.console.log(
1627                 f"Retry {retry_count}/{max_retries}: Failed to "
1628                 "generate fold prompts. Retrying..."
1629             )
1630
1631         for batch_size in batch_sizes:
1632             for fold_idx, fold_prompt in enumerate(fold_prompts):
1633                 plan = op_config.copy()
1634                 plan["fold_prompt"] = fold_prompt
1635                 plan["fold_batch_size"] = batch_size
1636                 plan["associative"] = is_associative
1637                 plan["name"] = f"

```

```

1662
1663     Returns:
1664         float: The calculated compression ratio.
1665     """
1666     reduce_key = op_config["reduce_key"]
1667     input_schema = op_config.get("input", {}).get("schema", {})
1668     output_schema = op_config["output"]["schema"]
1669     model = op_config.get("model", "gpt-4o-mini")
1670
1671     compression_ratios = []
1672
1673     # Handle both single key and list of keys
1674     if isinstance(reduce_key, list):
1675         distinct_keys = set(
1676             tuple(item[k] for k in reduce_key) for item in
1677             sample_input
1678         )
1679     else:
1680         distinct_keys = set(item[reduce_key] for item in
1681             sample_input)
1682
1683     for key in distinct_keys:
1684         if isinstance(reduce_key, list):
1685             key_input = [
1686                 item
1687                 for item in sample_input
1688                 if tuple(item[k] for k in reduce_key) == key
1689             ]
1690             key_output = [
1691                 item
1692                 for item in sample_output
1693                 if tuple(item[k] for k in reduce_key) == key
1694             ]
1695         else:
1696             key_input = [item for item in sample_input if
1697             item[reduce_key] == key]
1698             key_output = [item for item in sample_output if
1699             item[reduce_key] == key]
1700
1701         if input_schema:
1702             key_input_tokens = sum(
1703                 count_tokens(
1704                     json.dumps({k: item[k] for k in input_schema if
1705                         k in item}),
1706                         model,
1707                     )
1708                     for item in key_input
1709                 )
1710         else:
1711             key_input_tokens = sum(
1712                 count_tokens(json.dumps(item), model) for item in
1713                 key_input
1714             )
1715
1716         key_output_tokens = sum(
1717             count_tokens(
1718                 json.dumps({k: item[k] for k in output_schema if k
1719                     in item}), model
1720                     )
1721                     for item in key_output
1722                 )

```

```

1723
1724         compression_ratios[key] = (
1725             key_output_tokens / key_input_tokens if key_input_tokens
1726 > 0 else 1
1727         )
1728
1729     if not compression_ratios:
1730         return 1
1731
1732     # Calculate importance weights based on the number of items for
1733     # each key
1734     total_items = len(sample_input)
1735     if isinstance(reduce_key, list):
1736         importance_weights = {
1737             key: len(
1738                 [
1739                     item
1740                     for item in sample_input
1741                     if tuple(item[k] for k in reduce_key) == key
1742                 ]
1743             )
1744             / total_items
1745             for key in compression_ratios
1746         }
1747     else:
1748         importance_weights = {
1749             key: len([item for item in sample_input if
1750 item[reduce_key] == key])
1751             / total_items
1752             for key in compression_ratios
1753         }
1754
1755     # Calculate weighted average of compression ratios
1756     weighted_sum = sum(
1757         compression_ratios[key] * importance_weights[key]
1758         for key in compression_ratios
1759     )
1760     return weighted_sum
1761
1762     def _synthesize_fold_prompts(
1763         self,
1764         op_config: dict[str, Any],
1765         sample_input: list[dict[str, Any]],
1766         sample_output: list[dict[str, Any]],
1767         num_prompts: int = 2,
1768     ) -> list[str]:
1769         """
1770             Synthesize fold prompts for the reduce operation. We generate
1771             multiple
1772             fold prompts in case one is bad.
1773
1774             A fold operation is a higher-order function that iterates
1775             through a data structure,
1776             accumulating the results of applying a given combining operation
1777             to its elements.
1778             In the context of reduce operations, folding allows processing
1779             of data in batches,
1780             which can significantly improve performance for large datasets.
1781
1782             This method generates multiple fold prompts that can be used to
1783             optimize the reduce operation

```

```

1784     by allowing it to run on batches of inputs. It uses the language
1785     model to create prompts
1786     that are variations of the original reduce prompt, adapted for
1787     folding operations.
1788
1789     Args:
1790         op_config (dict[str, Any]): The configuration of the reduce
1791         operation.
1792         sample_input (list[dict[str, Any]]): A sample of the input
1793         data.
1794         sample_output (list[dict[str, Any]]): A sample of the output
1795         data.
1796         num_prompts (int, optional): The number of fold prompts to
1797         generate. Defaults to 2.
1798
1799     Returns:
1800         list[str]: A list of synthesized fold prompts.
1801
1802     The method performs the following steps:
1803     1. Sets up the system prompt and parameters for the language
1804     model.
1805     2. Defines a function to get random examples from the sample
1806     data.
1807     3. Creates a prompt template for generating fold prompts.
1808     4. Uses multi-threading to generate multiple fold prompts in
1809     parallel.
1810     5. Returns the list of generated fold prompts.
1811 """
1812     system_prompt = "You are an AI assistant tasked with creating a
1813     fold prompt for reduce operations in data processing pipelines."
1814     original_prompt = op_config["prompt"]
1815
1816     input_schema = op_config.get("input", {}).get("schema", {})
1817     output_schema = op_config["output"]["schema"]
1818
1819     def get_random_examples():
1820         reduce_key = op_config["reduce_key"]
1821         reduce_key = (
1822             list(reduce_key) if not isinstance(reduce_key, list)
1823         else reduce_key
1824             )
1825
1826         if reduce_key == ["_all"]:
1827             # For _all case, just pick random input and output
1828             examples
1829                 input_example = random.choice(sample_input)
1830                 output_example = random.choice(sample_output)
1831             elif isinstance(reduce_key, list):
1832                 random_key = tuple(
1833                     random.choice(
1834                         [
1835                             tuple(item[k] for k in reduce_key if k in
1836                             item)
1837
1838                             for item in sample_input
1839                             if all(k in item for k in reduce_key)
1840                         ]
1841                     )
1842                 input_example = random.choice(
1843                     [
1844                         item

```

```

1845             for item in sample_input
1846                 if all(item.get(k) == v for k, v in
1847                     zip(reduce_key, random_key))
1848                         ]
1849                     )
1850             output_example = random.choice(
1851                 [
1852                     item
1853                     for item in sample_output
1854                         if all(item.get(k) == v for k, v in
1855                             zip(reduce_key, random_key))
1856                                 ]
1857                             )
1858
1859             if input_schema:
1860                 input_example = {
1861                     k: input_example[k] for k in input_schema if k in
1862                     input_example
1863                         }
1864             output_example = {
1865                     k: output_example[k] for k in output_schema if k in
1866                     output_example
1867                         }
1868             return input_example, output_example
1869
1870         parameters = {
1871             "type": "object",
1872             "properties": {
1873                 "fold_prompt": {
1874                     "type": "string",
1875                         }
1876                     },
1877                     "required": ["fold_prompt"],
1878                 }
1879
1880     def generate_single_prompt():
1881         input_example, output_example = get_random_examples()
1882         prompt = f"""
1883             Original Reduce Operation Prompt:
1884             {original_prompt}
1885
1886             Sample Input:
1887             {json.dumps(input_example, indent=2)}
1888
1889             Sample Output:
1890             {json.dumps(output_example, indent=2)}
1891
1892             Create a fold prompt for the reduce operation to run on
1893             batches of inputs. The fold prompt should:
1894                 1. Minimally modify the original reduce prompt
1895                 2. Describe how to combine the new values with the current
1896                 reduced value
1897                 3. Be designed to work iteratively, allowing for multiple
1898                 fold operations. The first iteration will use the original prompt, and
1899                 all successive iterations will use the fold prompt.
1900
1901             The fold prompt should be a Jinja2 template with the
1902             following variables available:
1903                 - {{{ output }}}: The current reduced value (a dictionary
1904                 with the current output schema)
1905                 - {{{ inputs }}}: A list of new values to be folded in

```

```

1906             - {{{ reduce_key }}}}: The key used for grouping in the
1907             reduce operation
1908
1909             Provide the fold prompt as a string.
1910             """
1911             response = self.llm_client.generate_rewrite(
1912                 [{"role": "user", "content": prompt}],
1913                 system_prompt,
1914                 parameters,
1915             )
1916             fold_prompt =
1917             json.loads(response.choices[0].message.content)["fold_prompt"]

                # Run the operation with the fold prompt
                # Create a temporary plan with the fold prompt
                temp_plan = op_config.copy()
                temp_plan["fold_prompt"] = fold_prompt
                temp_plan["fold_batch_size"] = min(
                    len(sample_input), 2
                ) # Use a small batch size for testing

                # Run the operation with the fold prompt
                try:
                    self._run_operation(
                        temp_plan, sample_input[:temp_plan["fold_batch_size"]]
                    )

                    return fold_prompt
                except Exception as e:
                    self.console.log(
                        f"[red]Error in agent-generated fold prompt: {e}"
                    )
                
```

[/red]

```

                # Create a default fold prompt that instructs folding
                new data into existing output
                fold_prompt = f"""Analyze this batch of data using the
                following instructions:

                {original_prompt}

However, instead of starting fresh, fold your analysis into the existing
output that has already been generated. The existing output is provided
in the 'output' variable below:
```

{{{ output }}}}

Remember, you must fold the new data into the existing output, do not
start fresh."""

```

                    return fold_prompt

                    with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
                    fold_prompts = list(
                        executor.map(lambda _: generate_single_prompt(),
range(num_prompts))
                    )

                    return fold_prompts

```

```

def _evaluate_reduce_plans(
    self,
    op_config: dict[str, Any],
    plans: list[dict[str, Any]],
    input_data: list[dict[str, Any]],
    validator_prompt: str,
) -> dict[str, Any]:
    """
    Evaluate multiple reduce plans and select the best one.

    This method takes a list of reduce plans, evaluates each one
    using the input data
    and a validator prompt, and selects the best plan based on the
    evaluation scores.
    It also attempts to create and evaluate a merged plan that
    enhances the runtime performance
    of the best plan.

    A merged plan is an optimization technique applied to the best-
    performing plan
    that uses the fold operation. It allows the best plan to run
    even faster by
    executing parallel folds and then merging the results of these
    individual folds
    together. We default to a merge batch size of 2, but one can
    increase this.

    Args:
        op_config (dict[str, Any]): The configuration of the reduce
            operation.
        plans (list[dict[str, Any]]): A list of reduce plans to
            evaluate.
        input_data (list[dict[str, Any]]): The input data to use for
            evaluation.
        validator_prompt (str): The prompt to use for validating the
            output of each plan.

    Returns:
        dict[str, Any]: The best reduce plan, either the top-
            performing original plan
            or a merged plan if it performs well enough.

    The method performs the following steps:
    1. Evaluates each plan using multi-threading.
    2. Sorts the plans based on their evaluation scores.
    3. Selects the best plan and attempts to create a merged plan.
    4. Evaluates the merged plan and compares it to the best
    original plan.
    5. Returns either the merged plan or the best original plan
    based on their scores.
    """
    self.console.log("\nEvaluating Reduce Plans:")
    for i, plan in enumerate(plans):
        self.console.log(f"Plan {i+1} (batch size:
{plan['fold_batch_size']}")

    plan_scores = []
    plan_outputs = []

    # Create a fixed random sample for evaluation
    sample_size = min(100, len(input_data))

```

```

        evaluation_sample = random.sample(input_data, sample_size)

        # Create a fixed set of validation samples
        validation_inputs = self._create_validation_inputs(
            evaluation_sample, plan["reduce_key"]
        )

        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            futures = [
                executor.submit(
                    self._evaluate_single_plan,
                    plan,
                    evaluation_sample,
                    validator_prompt,
                    validation_inputs,
                )
                for plan in plans
            ]
            for future in as_completed(futures):
                plan, score, output = future.result()
                plan_scores.append((plan, score))
                plan_outputs[id(plan)] = output

            # Sort plans by score in descending order, then by
            fold_batch_size in descending order
            sorted_plans = sorted(
                plan_scores, key=lambda x: (x[1], x[0]["fold_batch_size"]),
                reverse=True
            )

            self.console.log("\n[bold]Reduce Plan Scores:[/bold]")
            for i, (plan, score) in enumerate(sorted_plans):
                self.console.log(
                    f"Plan {i+1} (batch size: {plan['fold_batch_size']}):
{score:.2f}"
                )

            best_plan, best_score = sorted_plans[0]
            self.console.log(
                f"\n[green]Selected best plan with score: {best_score:.2f}
and batch size: {best_plan['fold_batch_size']}[/green]"
            )

        if op_config.get("synthesize_merge", False):
            # Create a new plan with merge prompt and updated parameters
            merged_plan = best_plan.copy()

            # Synthesize merge prompt if it doesn't exist
            if "merge_prompt" not in merged_plan:
                merged_plan["merge_prompt"] =
self._synthesize_merge_prompt(
                    merged_plan, plan_outputs[id(best_plan)]
                )
            # Print the synthesized merge prompt
            self.console.log("\n[bold]Synthesized Merge Prompt:
[/bold]")
            self.console.log(merged_plan["merge_prompt"])

        # Set merge_batch_size to 2 and num_parallel_folds to 5
        merged_plan["merge_batch_size"] = 2
    
```

```

        # Evaluate the merged plan
        _, merged_plan_score, _, operation_instance =
self._evaluate_single_plan(
    merged_plan,
    evaluation_sample,
    validator_prompt,
    validation_inputs,
    return_instance=True,
)

# Get the merge and fold times from the operation instance
merge_times = operation_instance.merge_times
fold_times = operation_instance.fold_times
merge_avg_time = mean(merge_times) if merge_times else None
fold_avg_time = mean(fold_times) if fold_times else None

self.console.log("\n[bold]Scores:[/bold]")
self.console.log(f"Original plan: {best_score:.2f}")
self.console.log(f"Merged plan: {merged_plan_score:.2f}")

# Compare scores and decide which plan to use
if merged_plan_score >= best_score * 0.75:
    self.console.log(
        f"\n[green]Using merged plan with score:
{merged_plan_score:.2f}[/green]"
    )
    if merge_avg_time and fold_avg_time:
        merged_plan["merge_time"] = merge_avg_time
        merged_plan["fold_time"] = fold_avg_time
        return merged_plan
    else:
        self.console.log(
            f"\n[yellow]Merged plan quality too low. Using
original plan with score: {best_score:.2f}[/yellow]"
        )
        return best_plan
else:
    return best_plan

def _evaluate_single_plan(
    self,
    plan: dict[str, Any],
    input_data: list[dict[str, Any]],
    validator_prompt: str,
    validation_inputs: list[dict[str, Any]],
    return_instance: bool = False,
) -> (
    tuple[dict[str, Any], float, list[dict[str, Any]]]
    | tuple[dict[str, Any], float, list[dict[str, Any]]],
BaseOperation]
):
    """
    Evaluate a single reduce plan using the provided input data and
    validator prompt.

    This method runs the reduce operation with the given plan,
    validates the output,
    and calculates a score based on the validation results. The
    scoring works as follows:
    1. It counts the number of valid results from the validation.
    """

    # Evaluate the plan
    result = self._reduce(plan, input_data, validator_prompt, validation_inputs)
    # Validate the result
    validation_results = validate(result, validation_inputs)
    # Calculate the score
    score = calculate_score(validation_results)
    # Return the result and score
    if return_instance:
        return result, score
    else:
        return score

```

2. The score is calculated as the ratio of valid results to the total number of validation results.

3. This produces a score between 0 and 1, where 1 indicates all results were valid, and 0 indicates none were valid.

TODO: We should come up with a better scoring method here, maybe pairwise comparisons.

Args:

plan (dict[str, Any]): The reduce plan to evaluate.

input_data (list[dict[str, Any]]): The input data to use for evaluation.

validator_prompt (str): The prompt to use for validating the output.

return_instance (bool, optional): Whether to return the operation instance. Defaults to False.

Returns:

tuple[

tuple[dict[str, Any], float, list[dict[str, Any]]],
tuple[dict[str, Any], float, list[dict[str, Any]]],

BaseOperation],

]: A tuple containing the plan, its score, the output data, and optionally the operation instance.

The method performs the following steps:

1. Runs the reduce operation with the given plan on the input data.

2. Validates the output using the validator prompt.

3. Calculates a score based on the validation results.

4. Returns the plan, score, output data, and optionally the operation instance.

"""

```
output = self._run_operation(plan, input_data, return_instance)
```

```
if return_instance:
```

```
    output, operation_instance = output
```

```
validation_result = self._validate_reduce_output(
```

```
    plan, validation_inputs, output, validator_prompt
```

```
)
```

Calculate a score based on validation results

```
valid_count = sum(
```

```
1
```

```
for result in validation_result["validation_results"]
```

```
if result["is_correct"]
```

```
)
```

```
score = valid_count /
```

```
len(validation_result["validation_results"]))
```

```
if return_instance:
```

```
    return plan, score, output, operation_instance
```

```
else:
```

```
    return plan, score, output
```

```
def _synthesize_merge_prompt(
```

```
    self, plan: dict[str, Any], sample_outputs: list[dict[str, Any]]
```

```
) -> str:
```

```
"""
```

Synthesize a merge prompt for combining multiple folded outputs in a reduce operation.

This method generates a merge prompt that can be used to combine the results of multiple parallel fold operations into a single output. It uses the language model to create a prompt that is consistent with the original reduce and fold prompts while addressing the specific requirements of merging multiple outputs.

Args:

`plan (dict[str, Any]):` The reduce plan containing the original prompt and fold prompt.
`sample_outputs (list[dict[str, Any]]):` Sample outputs from the fold operation to use as examples.

Returns:

`str:` The synthesized merge prompt as a string.

The method performs the following steps:

1. Sets up the system prompt for the language model.
 2. Prepares a random sample output to use as an example.
 3. Creates a detailed prompt for the language model, including the original reduce prompt, fold prompt, sample output, and instructions for creating the merge prompt.
 4. Uses the language model to generate the merge prompt.
 5. Returns the generated merge prompt.
- """

```
system_prompt = "You are an AI assistant tasked with creating a merge prompt for reduce operations in data processing pipelines. The pipeline has a reduce operation, and incrementally folds inputs into a single output. We want to optimize the pipeline for speed by running multiple folds on different inputs in parallel, and then merging the fold outputs into a single output."
```

```
output_schema = plan["output"]["schema"]
random_output = random.choice(sample_outputs)
random_output = {
    k: random_output[k] for k in output_schema if k in
random_output
}
```

```
prompt = f"""Reduce Operation Prompt (runs on the first batch of
inputs):
{plan["prompt"]}
```

```
Fold Prompt (runs on the second and subsequent batches of
inputs):
{plan["fold_prompt"]}
```

```
Sample output of the fold operation (an input to the merge
operation):
{json.dumps(random_output, indent=2)}
```

Create a merge prompt for the reduce operation to combine 2+ folded outputs. The merge prompt should:

1. Give context on the task & fold operations, describing that the prompt will be used to combine multiple outputs from the fold operation (as if the original prompt was run on all inputs at once)
2. Describe how to combine multiple folded outputs into a single output

3. Minimally deviate from the reduce and fold prompts

The merge prompt should be a Jinja2 template with the following variables available:

- `outputs`: A list of reduced outputs to be merged (each following the output schema). You can access the first output with `outputs[0]` and the second with `outputs[1]`

Output Schema:

```
[json.dumps(output_schema, indent=2)]
```

Provide the merge prompt as a string.

```
"""
```

```
parameters = {
    "type": "object",
    "properties": {
        "merge_prompt": {
            "type": "string",
        }
    },
    "required": ["merge_prompt"],
}

response = self.llm_client.generate_rewrite(
    [{"role": "user", "content": prompt}],
    system_prompt,
    parameters,
)
return json.loads(response.choices[0].message.content)
["merge_prompt"]
```

```
__init__(runner, run_operation, num_fold_prompts=1, num_samples_in_validation=10)
```

Initialize the ReduceOptimizer.

Parameters:

Name	Type	Description	Default
config	dict[str, Any]	Configuration dictionary for the optimizer.	<i>required</i>
console	Console	Rich console object for pretty printing.	<i>required</i>
llm_client	LLMClient	Client for interacting with a language model.	<i>required</i>

Name	Type	Description	Default
max_threads	int	Maximum number of threads to use for parallel processing.	<i>required</i>
run_operation	Callable	Function to run an operation.	<i>required</i>
num_fold_prompts	int	Number of fold prompts to generate. Defaults to 1.	1
num_samples_in_validation	int	Number of samples to use in validation. Defaults to 10.	10

Source code in `docetl/optimizers/reduce_optimizer.py`

```

36     def __init__(self,
37         runner,
38         run_operation: Callable,
39         num_fold_prompts: int = 1,
40         num_samples_in_validation: int = 10,
41     ):
42         """
43             Initialize the ReduceOptimizer.
44
45             Args:
46                 config (dict[str, Any]): Configuration dictionary for the
47                 optimizer.
48                 console (Console): Rich console object for pretty printing.
49                 llm_client (LLMClient): Client for interacting with a language
50                 model.
51                 max_threads (int): Maximum number of threads to use for parallel
52                 processing.
53                 run_operation (Callable): Function to run an operation.
54                 num_fold_prompts (int, optional): Number of fold prompts to
55                 generate. Defaults to 1.
56                 num_samples_in_validation (int, optional): Number of samples to
57                 use in validation. Defaults to 10.
58             """
59         self.runner = runner
60         self.config = self.runner.config
61         self.console = self.runner.console
62         self.llm_client = self.runner.optimizer.llm_client
63         self._run_operation = run_operation
64         self.max_threads = self.runner.max_threads
65         self.num_fold_prompts = num_fold_prompts
66         self.num_samples_in_validation = num_samples_in_validation
67         self.status = self.runner.status

```

`optimize(op_config, input_data, level=1)`

Optimize the reduce operation based on the given configuration and input data.

This method performs the following steps: 1. Run the original operation 2. Generate a validator prompt 3. Validate the output 4. If improvement is needed: a. Evaluate if decomposition is beneficial b. If decomposition is beneficial, recursively optimize each sub-operation c. If not, proceed with single operation optimization 5. Run the optimized operation(s)

Parameters:

Name	Type	Description	Default
<code>op_config</code>	<code>dict[str, Any]</code>	Configuration for the reduce operation.	<i>required</i>
<code>input_data</code>	<code>list[dict[str, Any]]</code>	Input data for the reduce operation.	<i>required</i>

Returns:

Type	Description
<code>list[dict[str, Any]]</code>	<code>tuple[list[dict[str, Any]], list[dict[str, Any]], float]</code> : A tuple containing the list of optimized configurations
<code>list[dict[str, Any]]</code>	and the list of outputs from the optimized operation(s), and the cost of the operation due to synthesizing any resolve operations.

Source code in `docetl/optimizers/reduce_optimizer.py`

```
159 |     def optimize(
160 |         self,
161 |         op_config: dict[str, Any],
162 |         input_data: list[dict[str, Any]],
163 |         level: int = 1,
164 |     ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
165 |         """
166 |             Optimize the reduce operation based on the given configuration and
167 |             input data.
168 |
169 |             This method performs the following steps:
170 |                 1. Run the original operation
171 |                 2. Generate a validator prompt
172 |                 3. Validate the output
173 |                 4. If improvement is needed:
174 |                     a. Evaluate if decomposition is beneficial
175 |                     b. If decomposition is beneficial, recursively optimize each sub-
176 |                         operation
177 |                         c. If not, proceed with single operation optimization
178 |                 5. Run the optimized operation(s)
179 |
180 |             Args:
181 |                 op_config (dict[str, Any]): Configuration for the reduce
182 |                     operation.
183 |                 input_data (list[dict[str, Any]]): Input data for the reduce
184 |                     operation.
185 |
186 |             Returns:
187 |                 tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A tuple
188 |                     containing the list of optimized configurations
189 |                         and the list of outputs from the optimized operation(s), and the
190 |                         cost of the operation due to synthesizing any resolve operations.
191 |                         """
192 |             (
193 |                 validation_results,
194 |                 prompt_tokens,
195 |                 model_input_context_length,
196 |                 model,
197 |                 validator_prompt,
198 |                 original_output,
199 |             ) = self.should_optimize_helper(op_config, input_data)
200 |
201 |             # add_map_op = False
202 |             if prompt_tokens * 2 > model_input_context_length:
203 |                 # add_map_op = True
204 |                 self.console.log(
205 |                     f"[yellow]Warning: The reduce prompt exceeds the token limit"
206 |                 for model in model.
207 |                     f"Token count: {prompt_tokens}, Limit:
208 | {model_input_context_length}. "
209 |                     f"Add a map operation to the pipeline.[/yellow]"
210 |                 )
211 |
212 |             # # Also query an agent to look at a sample of the inputs and see if
213 |             they think a map operation would be helpful
214 |             # preprocessing_steps = ""
215 |             # should_use_map, preprocessing_steps = self._should_use_map(
```

```

216     #     op_config, input_data
217     # )
218     # if should_use_map or add_map_op:
219     #     # Synthesize a map operation
220     #     map_prompt, map_output_schema = self._synthesize_map_operation(
221     #         op_config, preprocessing_steps, input_data
222     #     )
223     #     # Change the reduce operation prompt to use the map schema
224     #     new_reduce_prompt =
225     self._change_reduce_prompt_to_use_map_schema(
226     #         op_config["prompt"], map_output_schema
227     #     )
228     #     op_config["prompt"] = new_reduce_prompt
229
230     #     # Return unoptimized map and reduce operations
231     #     return [map_prompt, op_config], input_data, 0.0
232
233     # Print the validation results
234     self.console.log("[bold]Validation Results on Initial Sample:
235 [/bold]")
236     if validation_results["needs_improvement"] or self.config.get(
237         "optimizer_config", {}
238     ).get("force_decompose", False):
239         self.console.post_optimizer_rationale(
240             should_optimize=True,
241             rationale="\n".join(
242                 [
243                     f"Issues: {result['issues']} Suggestions:
244 {result['suggestions']}"
245                     for result in
246 validation_results["validation_results"]
247                 ]
248             ),
249             validator_prompt=validator_prompt,
250         )
251         self.console.log(
252             "\n".join(
253                 [
254                     f"Issues: {result['issues']} Suggestions:
255 {result['suggestions']}"
256                     for result in
257 validation_results["validation_results"]
258                 ]
259             )
260         )
261
262     # Step 3: Evaluate if decomposition is beneficial
263     decomposition_result = self._evaluate_decomposition(
264         op_config, input_data, level
265     )
266
267     if decomposition_result["should_decompose"]:
268         return self._optimize_decomposed_reduce(
269             decomposition_result, op_config, input_data, level
270         )
271
272     return self._optimize_single_reduce(op_config, input_data,
273     validator_prompt)
274     else:
275         self.console.log(f"No improvements identified;
276 {validation_results}.")

```

```
        self.console.post_optimizer_rationale(
            should_optimize=False,
            rationale="No improvements identified; no optimization
recommended.",
            validator_prompt=validator_prompt,
        )
    return [op_config], original_output, 0.0
```

```
docetl.optimizers.join_optimizer.JoinOptimizer
```

Source code in `docetl/optimizers/join_optimizer.py`

```
15  class JoinOptimizer:
16      def __init__(self,
17          runner,
18          op_config: dict[str, Any],
19          target_recall: float = 0.95,
20          sample_size: int = 500,
21          sampling_weight: float = 20,
22          agent_max_retries: int = 5,
23          estimated_selectivity: float | None = None,
24      ):
25          self.runner = runner
26          self.config = runner.config
27          self.op_config = op_config
28          self.llm_client = runner.optimizer.llm_client
29          self.max_threads = runner.max_threads
30          self.console = runner.console
31          self.target_recall = target_recall
32          self.sample_size = sample_size
33          self.sampling_weight = sampling_weight
34          self.agent_max_retries = agent_max_retries
35          self.estimated_selectivity = estimated_selectivity
36          self.console.log(f"Target Recall: {self.target_recall}")
37          self.status = self.runner.status
38          self.max_comparison_sampling_attempts = 5
39          self.synthesized_keys = []
40          # if self.estimated_selectivity is not None:
41          #     self.console.log(
42          #         f"[yellow]Using estimated selectivity of
{self.estimated_selectivity}[/yellow]"
43          #     )
44
45
46
47      def _analyze_map_prompt_categorization(self, map_prompt: str) ->
48          tuple[bool, str]:
49          """
50              Analyze the map prompt to determine if it's explicitly
51              categorical.
52
53              Args:
54                  map_prompt (str): The map prompt to analyze.
55
56              Returns:
57                  bool: True if the prompt is explicitly categorical, False
58          otherwise.
59          """
60          messages = [
61              {
62                  "role": "system",
63                  "content": "You are an AI assistant tasked with
64          analyzing prompts for data processing operations.",
65              },
66              {
67                  "role": "user",
68                  "content": f"""Analyze the following map operation
69          prompt and determine if it is explicitly categorical,
70                  meaning it details a specific set of possible outputs:
71              """
72          ]
```

```

72             {map_prompt}
73
74             Respond with 'Yes' if the prompt is explicitly
75             categorical, detailing a finite set of possible outputs.
76             Respond with 'No' if the prompt allows for open-ended or
77             non-categorical responses.
78             Provide a brief explanation for your decision."""",
79         },
80     ]
81
82     response = self.llm_client.generate_rewrite(
83         messages,
84         "You are an expert in analyzing natural language prompts for
85         data processing tasks.",
86         {
87             "type": "object",
88             "properties": {
89                 "is_categorical": {
90                     "type": "string",
91                     "enum": ["Yes", "No"],
92                     "description": "Whether the prompt is explicitly
93                     categorical",
94                 },
95                 "explanation": {
96                     "type": "string",
97                     "description": "Brief explanation for the
98                     decision",
99                 },
100            },
101        },
102    ),
103 )
104
105 analysis = json.loads(response.choices[0].message.content)
106
107 self.console.log("[bold]Map Prompt Analysis:[/bold]")
108 self.console.log(f"Is Categorical:
109 {analysis['is_categorical']}"))
110 self.console.log(f"Explanation: {analysis['explanation']}")

111
112     return analysis["is_categorical"].lower() == "yes",
113 analysis["explanation"]

114
115 def _determine_duplicate_keys(
116     self,
117     input_data: list[dict[str, Any]],
118     reduce_key: list[str],
119     map_prompt: str | None = None,
120 ) -> tuple[bool, str]:
121     # Prepare a sample of the input data for analysis
122     sample_size = min(10, len(input_data))
123     data_sample = random.sample(
124         [{rk: item[rk] for rk in reduce_key} for item in
125         input_data], sample_size
126     )
127
128     context_prefix = """
129     if map_prompt:
130         context_prefix = f"For context, these values came out of a
131         pipeline with the following prompt:\n\n{map_prompt}\n\n"

```

```

133     messages = [
134         {
135             "role": "user",
136             "content": f"{context_prefix}I want to do a reduce
137 operation on these values, and I need to determine if there are semantic
138 duplicates in the data, where the strings are different but they
139 technically belong in the same group. Note that exact string duplicates
140 should not be considered here.\n\nHere's a sample of the data (showing
141 the '{reduce_key}' field(s)): {data_sample}\n\nBased on this {'context'
142 and ' if map_prompt else ''}sample, are there likely to be such semantic
143 duplicates (not exact string matches) in the dataset? Respond with 'yes'
144 only if you think there are semantic duplicates, or 'no' if you don't
145 see evidence of semantic duplicates or if you only see exact string
146 duplicates.",
147         },
148     ]
149     response = self.llm_client.generate_rewrite(
150         messages,
151         "You are an expert data analyst. Analyze the given data
152 sample and determine if there are likely to be semantic duplicate values
153 that belong in the same group, even if the strings are different.",
154         {
155             "type": "object",
156             "properties": {
157                 "likely_duplicates": {
158                     "type": "string",
159                     "enum": ["Yes", "No"],
160                     "description": "Whether duplicates are likely to
161 exist in the full dataset",
162                 },
163                 "explanation": {
164                     "type": "string",
165                     "description": "Brief explanation for the
166 decision",
167                 },
168             },
169             "required": ["likely_duplicates", "explanation"],
170         },
171     )
172
173     analysis = json.loads(response.choices[0].message.content)
174
175     self.console.log(f"[bold]Duplicate Analysis for '{reduce_key}':"
176     [/bold]")
177     self.console.log(f"Likely Duplicates:
178 {analysis['likely_duplicates']}")
179     self.console.log(f"Explanation: {analysis['explanation']}")

180     if analysis["likely_duplicates"].lower() == "yes":
181         self.console.log(
182             "[yellow]Duplicates are likely. Consider using a
183 deduplication strategy in the resolution step.[/yellow]"
184         )
185     return True, analysis["explanation"]
186
187     return False, ""

188     def _sample_random_pairs(
189         self, input_data: list[dict[str, Any]], n: int
190     ) -> list[tuple[int, int]]:
191         """Sample random pairs of indices, excluding exact matches."""
192         pairs = set()

```

```

194         max_attempts = n * 10 # Avoid infinite loop
195         attempts = 0
196
197         while len(pairs) < n and attempts < max_attempts:
198             i, j = random.sample(range(len(input_data)), 2)
199             if i != j and input_data[i] != input_data[j]:
200                 pairs.add((min(i, j), max(i, j))) # Ensure ordered
201         pairs
202         attempts += 1
203
204     return list(pairs)
205
206     def _check_duplicates_with_llm(
207         self,
208         input_data: list[dict[str, Any]],
209         pairs: list[tuple[int, int]],
210         reduce_key: list[str],
211         map_prompt: str | None = None,
212     ) -> tuple[bool, str]:
213         """Use LLM to check if any pairs are duplicates."""
214
215         content = "Analyze the following pairs of entries and determine
216         if any of them are likely duplicates. Respond with 'Yes' if you find any
217         likely duplicates, or 'No' if none of the pairs seem to be duplicates.
218         Provide a brief explanation for your decision.\n\n"
219
220         if map_prompt:
221             content = (
222                 f"For reference, here is the map prompt used earlier in
223                 the pipeline: {map_prompt}\n\n"
224                 + content
225             )
226
227         for i, (idx1, idx2) in enumerate(pairs, 1):
228             content += f"Pair {i}:\n"
229             content += "Entry 1:\n"
230             for key in reduce_key:
231                 content += f"{key}: {json.dumps(input_data[idx1][key], indent=2)}\n"
232             content += "\nEntry 2:\n"
233             for key in reduce_key:
234                 content += f"{key}: {json.dumps(input_data[idx2][key], indent=2)}\n"
235             content += "\n"
236
237         messages = [{"role": "user", "content": content}]
238
239         system_prompt = "You are an AI assistant tasked with identifying
240         potential duplicate entries in a dataset."
241         response_schema = {
242             "type": "object",
243             "properties": {
244                 "duplicates_found": {"type": "string", "enum": ["Yes", "No"]},
245                 "explanation": {"type": "string"},
246             },
247             "required": ["duplicates_found", "explanation"],
248         }
249
250         response = self.llm_client.generate_rewrite(
251             messages, system_prompt, response_schema
252
253
254

```

```

255         )
256
257     # Print the duplicates_found and explanation
258     self.console.log(
259         f"[bold]Duplicates in keys found: [/bold]"
260     {response['duplicates_found']}\\n"
261         f"[bold]Explanation: [/bold] {response['explanation']}"
262     )
263
264     return response["duplicates_found"].lower() == "yes",
265     response["explanation"]
266
267     def synthesize_compare_prompt(
268         self, map_prompt: str | None, reduce_key: list[str]
269     ) -> str:
270
271         system_prompt = f"You are an AI assistant tasked with creating a
272         comparison prompt for LLM-assisted entity resolution. Your task is to
273         create a comparison prompt that will be used to compare two entities,
274         referred to as input1 and input2, to see if they are likely the same
275         entity based on the following reduce key(s): {', '.join(reduce_key)}."
276         if map_prompt:
277             system_prompt += f"\n\nFor context, here is the prompt used
278             earlier in the pipeline to create the inputs to resolve: {map_prompt}"
279
280         messages = [
281             {
282                 "role": "user",
283                 "content": f"""
284                 Create a comparison prompt for entity resolution: The prompt should:
285                 1. Be tailored to the specific domain and type of data being
286                     compared ({reduce_key}), based on the context provided.
287                 2. Instruct to compare two entities, referred to as input1 and
288                     input2.
289                 3. Specifically mention comparing each reduce key in input1 and
290                     input2 (e.g., input1.{key} and input2.{key} for each key in
291                     {reduce_key}). You can reference other fields in the input as well, as
292                     long as they are short.
293                 4. Include instructions to consider relevant attributes or
294                     characteristics for comparison.
295                 5. Ask to respond with "True" if the entities are likely the same,
296                     or "False" if they are likely different.
297
298             Example structure:
299             ```
300                 Compare the following two {reduce_key} from [entity or document
301                 type]:
302
303                 [Entity 1]:
304                 {{{{ input1.key1 }}}}
305                 {{{{ input1.optional_key2 }}}}
306
307                 [Entity 2]:
308                 {{{{ input2.key1 }}}}
309                 {{{{ input2.optional_key2 }}}}
310
311                 Are these [entities] likely referring to the same [entity type]?
312                 Consider [list relevant attributes or characteristics to compare].
313                 Respond with "True" if they are likely the same [entity type], or
314                 "False" if they are likely different [entity types].
315                 ```


```

```

316
317     Please generate the comparison prompt, which should be a Jinja2
318     template:
319     """
320     ]
321
322     response = self.llm_client.generate_rewrite(
323         messages,
324         system_prompt,
325         {
326             "type": "object",
327             "properties": {
328                 "comparison_prompt": {
329                     "type": "string",
330                     "description": "Detailed comparison prompt for
entity resolution",
331                     }
332                 },
333                 "required": ["comparison_prompt"],
334             },
335         )
336
337     comparison_prompt =
338 json.loads(response.choices[0].message.content)[
339     "comparison_prompt"
340     ]
341
342
343     # Log the synthesized comparison prompt
344     self.console.log("[green]Synthesized comparison prompt:
345 [ /green]")
346     self.console.log(comparison_prompt)
347
348     if not comparison_prompt:
349         raise ValueError(
350             "Could not synthesize a comparison prompt. Please
provide a comparison prompt in the config."
351         )
352
353
354     return comparison_prompt
355
356
357     def synthesize_resolution_prompt(
358         self,
359         map_prompt: str | None,
360         reduce_key: list[str],
361         output_schema: dict[str, str],
362     ) -> str:
363         system_prompt = f"""You are an AI assistant tasked with creating
364 a resolution prompt for LLM-assisted entity resolution.
365 Your task is to create a prompt that will be used to merge
366 multiple duplicate keys into a single, consolidated key.
367 The key(s) being resolved (known as the reduce_key) are ',',
368 '.join(reduce_key)}.
369 The duplicate keys will be provided in a list called 'inputs' in
370 a Jinja2 template.
371 """
372
373     if map_prompt:
374         system_prompt += f"\n\nFor context, here is the prompt used
375 earlier in the pipeline to create the inputs to resolve: {map_prompt}"
```

```

377     messages = [
378         {
379             "role": "user",
380             "content": f"""
381             Create a resolution prompt for merging duplicate keys into a single
382             key. The prompt should:
383                 1. Be tailored to the specific domain and type of data being merged,
384                 based on the context provided.
385                 2. Use a Jinja2 template to iterate over the duplicate keys
386                 (accessed as 'inputs', where each item is a dictionary containing the
387                 reduce_key fields, which you can access as entry.reduce_key for each
388                 reduce_key in {reduce_key}).
389                 3. Instruct to create a single, consolidated key from the duplicate
390                 keys.
391                 4. Include guidelines for resolving conflicts (e.g., choosing the
392                 most recent, most complete, or most reliable information).
393                 5. Specify that the output of the resolution prompt should conform
394                 to the given output schema: {json.dumps(output_schema, indent=2)}
395
396             Example structure:
397             ```
398             Analyze the following duplicate entries for the {reduce_key} key:
399
400             {{% for key in inputs %}}
401             Entry {{{{ loop.index }}}}: {{{
402                 % for key in reduce_key %}
403                 {{{{ key }}}}: {{{{ key[reduce_key] }}}}
404             }}%
405             {{% endfor %}}
406
407             {{% endfor %}}
408
409             Merge these into a single key.
410             When merging, follow these guidelines:
411                 1. [Provide specific merging instructions relevant to the data type]
412                 2. [Do not make the prompt too long]
413
414             Ensure that the merged key conforms to the following schema:
415             {json.dumps(output_schema, indent=2)}
416
417             Return the consolidated key as a single [appropriate data type]
418             value.
419             ```
420
421             Please generate the resolution prompt:
422             """
423             }
424
425             response = self.llm_client.generate_rewrite(
426                 messages,
427                 system_prompt,
428                 {
429                     "type": "object",
430                     "properties": {
431                         "resolution_prompt": {
432                             "type": "string",
433                             "description": "Detailed resolution prompt for
434                             merging duplicate keys",
435                             }
436                         },
437                         "required": ["resolution_prompt"],

```

```

438         },
439     )
440
441     resolution_prompt =
442 json.loads(response.choices[0].message.content)[
443         "resolution_prompt"
444     ]
445
446     # Log the synthesized resolution prompt
447     self.console.log("[green]Synthesized resolution prompt:
448 [/green]")
449     self.console.log(resolution_prompt)
450
451     if not resolution_prompt:
452         raise ValueError(
453             "Could not synthesize a resolution prompt. Please
454 provide a resolution prompt in the config."
455         )
456
457     return resolution_prompt
458
459     def should_optimize(self, input_data: list[dict[str, Any]]) ->
460 tuple[bool, str]:
461         """
462             Determine if the given operation configuration should be
463             optimized.
464             """
465             # If there are no blocking keys or embeddings, then we don't
466             need to optimize
467             if not self.op_config.get("blocking_conditions") or not
468 self.op_config.get(
469                 "blocking_threshold"
470             ):
471                 return True, ""
472
473             # Check if the operation is marked as empty
474             elif self.op_config.get("empty", False):
475                 # Extract the map prompt from the intermediates
476                 map_prompt = self.op_config["_intermediates"]["map_prompt"]
477                 reduce_key = self.op_config["_intermediates"]["reduce_key"]
478
479                 if reduce_key is None:
480                     raise ValueError(
481                         "[yellow]Warning: No reduce key found in
482                         intermediates for synthesized resolve operation.[/yellow]"
483                     )
484
485                 dedup = True
486                 explanation = "There is a reduce operation that does not
487 follow a resolve operation. Consider adding a resolve operation to
488 deduplicate the data."
489
490                 if map_prompt:
491                     # Analyze the map prompt
492                     analysis, explanation =
493 self._analyze_map_prompt_categorization(
494                         map_prompt
495                     )
496
497                     if analysis:
498                         dedup = False

```

```

499         else:
500             self.console.log(
501                 "[yellow]No map prompt found in intermediates for
502 analysis.[/yellow]"
503             )
504
505         # TODO: figure out why this would ever be the case
506         if not map_prompt:
507             map_prompt = "N/A"
508
509         if dedup is False:
510             dedup, explanation = self._determine_duplicate_keys(
511                 input_data, reduce_key, map_prompt
512             )
513
514         # Now do the last attempt of pairwise comparisons
515         if dedup is False:
516             # Sample up to 20 random pairs of keys for duplicate
517             analysis
518             sampled_pairs = self._sample_random_pairs(input_data,
519             20)
520
521             # Use LLM to check for duplicates
522             duplicates_found, explanation =
523             self._check_duplicates_with_llm(
524                 input_data, sampled_pairs, reduce_key, map_prompt
525             )
526
527             if duplicates_found:
528                 dedup = True
529
530         return dedup, explanation
531
532     return False, ""
533
534     def optimize_resolve(
535         self, input_data: list[dict[str, Any]]
536     ) -> tuple[dict[str, Any], float]:
537         # Check if the operation is marked as empty
538         if self.op_config.get("empty", False):
539             # Extract the map prompt from the intermediates
540             dedup, _ = self.should_optimize(input_data)
541             reduce_key = self.op_config["_intermediates"]["reduce_key"]
542             map_prompt = self.op_config["_intermediates"]["map_prompt"]
543
544         if dedup is False:
545             # If no deduplication is needed, return the same config
546             with 0 cost
547                 return self.op_config, 0.0
548
549             # Add the reduce key to the output schema in the config
550             self.op_config["output"] = {"schema": {rk: "string" for rk
551             in reduce_key}}
552             for attempt in range(2): # Try up to 2 times
553                 self.op_config["comparison_prompt"] =
554                 self.synthesize_compare_prompt(
555                     map_prompt, reduce_key
556                 )
557                 if (
558                     "input1" in self.op_config["comparison_prompt"]
559                     and "input2" in self.op_config["comparison_prompt"]

```

```

560         ):
561             break
562         elif attempt == 0:
563             self.console.log(
564                 "[yellow]Warning: 'input1' or 'input2' not found
565                 in comparison prompt. Retrying...[/yellow]"
566             )
567         if (
568             "input1" not in self.op_config["comparison_prompt"]
569             or "input2" not in self.op_config["comparison_prompt"]
570         ):
571             self.console.log(
572                 "[red]Error: Failed to generate comparison prompt
573                 with 'input1' and 'input2'. Using last generated prompt.[/red]"
574             )
575             for attempt in range(2): # Try up to 2 times
576                 self.op_config["resolution_prompt"] =
577                     self.synthesize_resolution_prompt(
578                         map_prompt, reduce_key, self.op_config["output"]
579                         ["schema"])
580                     )
581             if "inputs" in self.op_config["resolution_prompt"]:
582                 break
583             elif attempt == 0:
584                 self.console.log(
585                     "[yellow]Warning: 'inputs' not found in
586                     resolution prompt. Retrying...[/yellow]"
587                 )
588             if "inputs" not in self.op_config["resolution_prompt"]:
589                 self.console.log(
590                     "[red]Error: Failed to generate resolution prompt
591                     with 'inputs'. Using last generated prompt.[/red]"
592                 )
593
594             # Pop off the empty flag
595             self.op_config.pop("empty")
596
597             embeddings, blocking_keys, embedding_cost =
598                 self._compute_embeddings(input_data)
599             self.console.log(
600                 f"[bold]Cost of creating embeddings on the sample:
601 ${embedding_cost:.4f}[/bold]"
602             )
603
604             similarities = self._calculate_cosine_similarities(embeddings)
605
606             sampled_pairs = self._sample_pairs(similarities)
607             comparison_results, comparison_cost =
608                 self._perform_comparisons_resolve(
609                     input_data, sampled_pairs
610                 )
611
612             self._print_similarity_histogram(similarities,
613             comparison_results)
614
615             threshold, estimated_selectivity =
616                 self._find_optimal_threshold(
617                     comparison_results, similarities
618                 )
619
620             blocking_rules = self._generate_blocking_rules(
621                 blocking_keys, input_data, comparison_results

```

```

621         )
622
623     if blocking_rules:
624         false_negatives, rule_selectivity =
625     self._verify_blocking_rule(
626         input_data,
627         blocking_rules[0],
628         blocking_keys,
629         comparison_results,
630     )
631     # If more than 50% of the sample is false negatives, reject
632     # the blocking rule
633     if len(false_negatives) > len(sampled_pairs) / 2:
634         if false_negatives:
635             self.console.log(
636                 f"[red]Blocking rule rejected.
637 {len(false_negatives)} false negatives detected in the sample
638 ({len(false_negatives)} / len(sampled_pairs)::2f} of the sample).[/red]"
639             )
640         for i, j in false_negatives[:5]: # Show up to 5
641             examples
642                 self.console.log(
643                     f" Filtered pair: {{ {blocking_keys[0]}:
644 {input_data[i][blocking_keys[0]]} }} and {{ {blocking_keys[0]}:
645 {input_data[j][blocking_keys[0]]} }}"
646                     )
647         if len(false_negatives) > 5:
648             self.console.log(f" ... and
649 {len(false_negatives) - 5} more.")
650         blocking_rules = (
651             []
652         ) # Clear the blocking rule if it introduces false
653         negatives or is too selective
654         elif not false_negatives and rule_selectivity >
655         estimated_selectivity:
656             self.console.log(
657                 "[green]Blocking rule verified. No false negatives
658 detected in the sample and selectivity is within estimated selectivity.
659 [/green]"
660             )
661         else:
662             # TODO: ask user if they want to use the blocking rule,
663             # or come up with some good default behavior
664             blocking_rules = []
665
666         optimized_config = self._update_config(threshold, blocking_keys,
667         blocking_rules)
668         return optimized_config, embedding_cost + comparison_cost
669
670     def optimize_equijoin(
671         self,
672         left_data: list[dict[str, Any]],
673         right_data: list[dict[str, Any]],
674         skip_map_gen: bool = False,
675         skip_containment_gen: bool = False,
676     ) -> tuple[dict[str, Any], float, dict[str, Any]]:
677         left_keys = self.op_config.get("blocking_keys", {}).get("left",
678         [])
679         right_keys = self.op_config.get("blocking_keys",
680         {}).get("right", [])

```

```

682         if not left_keys and not right_keys:
683             # Ask the LLM agent if it would be beneficial to do a map
684             operation on
685                 # one of the datasets before doing an equijoin
686                 apply_transformation, dataset_to_transform, reason = (
687                     (
688                         self._should_apply_map_transformation(
689                             left_keys, right_keys, left_data, right_data
690                         )
691                     )
692                     if not skip_map_gen
693                     else (False, None, None)
694                 )
695
696             if apply_transformation and not skip_map_gen:
697                 self.console.log(
698                     f"LLM agent suggested applying a map transformation
699                     to {dataset_to_transform} dataset because: {reason}"
700                     )
701                     extraction_prompt, output_key, new_comparison_prompt = (
702                         self._generate_map_and_new_join_transformation(
703                             dataset_to_transform, reason, left_data,
704                             right_data
705                             )
706                         )
707                         self.console.log(
708                             f"Generated map transformation prompt:
709                             {extraction_prompt}"
710                             )
711                             self.console.log(f"\nNew output key: {output_key}")
712                             self.console.log(
713                                 f"\nNew equijoin comparison prompt:
714                                 {new_comparison_prompt}"
715                             )
716
717             # Update the comparison prompt
718             self.op_config["comparison_prompt"] =
719             new_comparison_prompt
720
721             # Add the output key to the left_keys or right_keys
722             if dataset_to_transform == "left":
723                 left_keys.append(output_key)
724             else:
725                 right_keys.append(output_key)
726
727             # Reset the blocking keys in the config
728             self.op_config["blocking_keys"] = {
729                 "left": left_keys,
730                 "right": right_keys,
731             }
732
733             # Bubble up this config and return the transformation
734             prompt, so we can optimize the map operation
735             return (
736                 self.op_config,
737                 0.0,
738                 {
739                     "optimize_map": True,
740                     "map_prompt": extraction_prompt,
741                     "output_key": output_key,
742                     "dataset_to_transform": dataset_to_transform,

```

```

743             },
744         )
745
746         # Print the reason for not applying a map transformation
747         self.console.log(
748             f"Reason for not synthesizing a map transformation for
749             either left or right dataset: {reason}"
750         )
751
752         # If there are no blocking keys, generate them
753         if not left_keys or not right_keys:
754             generated_left_keys, generated_right_keys = (
755                 self._generate_blocking_keys_equijoin(left_data,
756             right_data)
757             )
758             left_keys.extend(generated_left_keys)
759             right_keys.extend(generated_right_keys)
760             left_keys = list(set(left_keys))
761             right_keys = list(set(right_keys))
762
763             # Log the generated blocking keys
764             self.console.log(
765                 "[bold]Generated blocking keys (for embeddings-based
766             blocking):[/bold]"
767             )
768             self.console.log(f"Left keys: {left_keys}")
769             self.console.log(f"Right keys: {right_keys}")
770
771             left_embeddings, _, left_embedding_cost =
772             self._compute_embeddings(
773                 left_data, keys=left_keys
774             )
775             right_embeddings, _, right_embedding_cost =
776             self._compute_embeddings(
777                 right_data, keys=right_keys
778             )
779             self.console.log(
780                 f"[bold]Cost of creating embeddings on the sample:
781             ${left_embedding_cost + right_embedding_cost:.4f}[/bold]"
782             )
783
784             similarities = self._calculate_cross_similarities(
785                 left_embeddings, right_embeddings
786             )
787
788             sampled_pairs = self._sample_pairs(similarities)
789             comparison_results, comparison_cost =
790             self._perform_comparisons_equijoin(
791                 left_data, right_data, sampled_pairs
792             )
793             self._print_similarity_histogram(similarities,
794             comparison_results)
795             attempts = 0
796             while (
797                 not any(result[2] for result in comparison_results)
798                 and attempts < self.max_comparison_sampling_attempts
799             ):
800                 self.console.log(
801                     "[yellow]No matches found in the current sample.
802             Resampling pairs to compare...[/yellow]"
803                 )

```

```

804         sampled_pairs = self._sample_pairs(similarities)
805         comparison_results, current_cost =
806     self._perform_comparisons_equijoin(
807         left_data, right_data, sampled_pairs
808     )
809     comparison_cost += current_cost
810     self._print_similarity_histogram(similarities,
811 comparison_results)
812     attempts += 1
813
814     if not any(result[2] for result in comparison_results):
815         # If still no matches after max_comparison_sampling_attempts
816         attempts, use 99th percentile similarity as threshold
817         # This is a heuristic to avoid being in an infinite loop
818         # TODO: have a better plan for sampling pairs or avoiding
819         getting into this situation
820         self.console.log(
821             f"[yellow]No matches found after
822 {self.max_comparison_sampling_attempts} attempts. Using 99th percentile
823 similarity as threshold.[/yellow]"
824         )
825         threshold = np.percentile([sim[2] for sim in similarities],
826 99)
827         # TODO: figure out how to estimate selectivity
828         estimated_selectivity = 0.0
829         self.estimated_selectivity = estimated_selectivity
830
831     else:
832         threshold, estimated_selectivity =
833     self._find_optimal_threshold(
834         comparison_results, similarities
835     )
836     self.estimated_selectivity = estimated_selectivity
837
838     blocking_rules = self._generate_blocking_rules_equijoin(
839         left_keys, right_keys, left_data, right_data,
840 comparison_results
841     )
842
843     if blocking_rules:
844         false_negatives, rule_selectivity =
845     self._verify_blocking_rule_equijoin(
846         left_data,
847         right_data,
848         blocking_rules[0],
849         left_keys,
850         right_keys,
851         comparison_results,
852     )
853         if not false_negatives and rule_selectivity <=
854 estimated_selectivity:
855             self.console.log(
856                 "[green]Blocking rule verified. No false negatives
857 detected in the sample and selectivity is within bounds.[/green]"
858             )
859     else:
860         if false_negatives:
861             self.console.log(
862                 f"[red]Blocking rule rejected.
863 {len(false_negatives)} false negatives detected in the sample.[/red]"
864             )

```

```

865             for i, j in false_negatives[:5]: # Show up to 5
866     examples
867             self.console.log(
868                 f" Filtered pair: Left: {{'{
869 '.join(f'{key}: {left_data[i][key]}' for key in left_keys)}}} and Right:
870 {{' ', '.join(f'{key}: {right_data[j][key]}' for key in right_keys)}}}"
871                     )
872             if len(false_negatives) > 5:
873                 self.console.log(f" ... and
874 {len(false_negatives) - 5} more.")
875             if rule_selectivity > estimated_selectivity:
876                 self.console.log(
877                     f"[red]Blocking rule rejected. Rule selectivity
878 ({rule_selectivity:.4f}) is higher than the estimated selectivity
879 ({estimated_selectivity:.4f}).[/red]"
880                     )
881             blocking_rules = (
882                 []
883             ) # Clear the blocking rule if it introduces false
884             negatives or is too selective
885
886             containment_rules = self._generateContainmentRulesEquijoin(
887                 left_data, right_data
888             )
889             if not skipContainmentGen:
890                 self.console.log(
891                     f"[bold]Generated {len(containment_rules)} containment
892 rules. Please select which ones to use as blocking conditions:[/bold]"
893                     )
894             selectedContainmentRules = []
895             for rule in containment_rules:
896                 self.console.log(f"[green]{rule}[/green]")
897                 # Temporarily stop the status
898                 if self.status:
899                     self.status.stop()
900                     # Use Rich's Confirm for input
901                     if Confirm.ask("Use this rule?", console=self.console):
902                         selectedContainmentRules.append(rule)
903                         # Restart the status
904                         if self.status:
905                             self.status.start()
906             else:
907                 # Take first 2
908                 selectedContainmentRules = containment_rules[:2]
909
910             if len(containment_rules) > 0:
911                 self.console.log(
912                     f"[bold]Selected {len(selectedContainmentRules)}"
913                     containment rules for blocking.[/bold]"
914                     )
915             blocking_rules.extend(selectedContainmentRules)
916
917             optimizedConfig = self._updateConfigEquijoin(
918                 threshold, left_keys, right_keys, blocking_rules
919             )
920             return (
921                 optimizedConfig,
922                 left_embedding_cost + right_embedding_cost +
923                 comparison_cost,
924                 {},
925             )

```

```

926
927     def _should_apply_map_transformation(
928         self,
929         left_keys: list[str],
930         right_keys: list[str],
931         left_data: list[dict[str, Any]],
932         right_data: list[dict[str, Any]],
933         sample_size: int = 5,
934     ) -> tuple[bool, str, str]:
935         # Sample data
936         left_sample = random.sample(left_data, min(sample_size,
937             len(left_data)))
938         right_sample = random.sample(right_data, min(sample_size,
939             len(right_data)))
940
941         # Get keys and their average lengths
942         all_left_keys = {
943             k: sum(len(str(d[k])) for d in left_sample) /
944             len(left_sample)
945             for k in left_sample[0].keys()
946         }
947         all_right_keys = {
948             k: sum(len(str(d[k])) for d in right_sample) /
949             len(right_sample)
950             for k in right_sample[0].keys()
951         }
952
953         messages = [
954             {
955                 "role": "user",
956                 "content": f"""Analyze the following datasets and
957 determine if an additional LLM transformation should be applied to
958 generate a new key-value pair for easier joining:
959
960                 Comparison prompt for the join operation:
961 {self.op_config.get('comparison_prompt', 'No comparison prompt
962 provided.')}
963
964                 Left dataset keys and average lengths:
965 {json.dumps(all_left_keys, indent=2)}
966                 Right dataset keys and average lengths:
967 {json.dumps(all_right_keys, indent=2)}
968
969                 Left dataset sample:
970 {json.dumps(left_sample, indent=2)}
971
972                 Right dataset sample:
973 {json.dumps(right_sample, indent=2)}
974
975                 Current keys used for embedding-based ranking of likely
976 matches:
977                 Left keys: {left_keys}
978                 Right keys: {right_keys}
979
980                 Consider the following:
981                 1. Are the current keys sufficient for accurate
982 embedding-based ranking of likely matches? We don't want to use too many
983 keys, or keys with too much information, as this will dilute the signal
984 in the embeddings.
985                 2. Are there any keys particularly long (e.g., full text
986 fields), containing information that is not relevant for the join

```

```

987     operation? The dataset with the longer keys should be transformed.
988         3. Would a summary or extraction of important
989             information from long key-value pairs be beneficial? If so, the dataset
990             with the longer keys should be transformed.
991         4. Is there a mismatch in information representation
992             between the datasets?
993         5. Could an additional LLM-generated field improve the
994             accuracy of embeddings or join comparisons?
995
996         If you believe an additional LLM transformation would be
997             beneficial, specify which dataset (left or right) should be transformed
998             and explain why. Otherwise, indicate that no additional transformation
999             is needed and explain why the current blocking keys are sufficient.""",
1000     }
1001 ]
1002
1003     response = self.llm_client.generate_rewrite(
1004         messages,
1005         "You are an AI expert in data analysis and entity
1006         matching.",
1007         {
1008             "type": "object",
1009             "properties": {
1010                 "apply_transformation": {"type": "boolean"},
1011                 "dataset_to_transform": {
1012                     "type": "string",
1013                     "enum": ["left", "right", "none"],
1014                 },
1015                 "reason": {"type": "string"},
1016             },
1017             "required": ["apply_transformation",
1018             "dataset_to_transform", "reason"],
1019         },
1020     )
1021
1022     result = json.loads(response.choices[0].message.content)
1023
1024     return (
1025         result["apply_transformation"],
1026         result["dataset_to_transform"],
1027         result["reason"],
1028     )
1029
1030     def _generate_map_and_new_join_transformation(
1031         self,
1032         dataset_to_transform: str,
1033         reason: str,
1034         left_data: list[dict[str, Any]],
1035         right_data: list[dict[str, Any]],
1036         sample_size: int = 5,
1037     ) -> tuple[str, str, str]:
1038         # Sample data
1039         left_sample = random.sample(left_data, min(sample_size,
1040             len(left_data)))
1041         right_sample = random.sample(right_data, min(sample_size,
1042             len(right_data)))
1043
1044         target_data = left_sample if dataset_to_transform == "left" else
1045         right_sample
1046
1047         messages = [

```

```

1048         {
1049             "role": "user",
1050             "content": f"""Generate an LLM prompt to transform the
1051 {dataset_to_transform} dataset for easier joining. The transformation
1052 should create a new key-value pair.
1053
1054             Current comparison prompt for the join operation:
1055 {self.op_config.get('comparison_prompt', 'No comparison prompt
1056 provided.')}
1057
1058             Target ({dataset_to_transform}) dataset sample:
1059 {json.dumps(target_data, indent=2)}
1060
1061             Other ('left' if dataset_to_transform == "right" else
1062 "right") dataset sample:
1063 {json.dumps(right_sample if dataset_to_transform ==
1064 "left" else left_sample, indent=2)}
1065
1066             Reason for transforming {dataset_to_transform} dataset:
1067 {reason}
1068
1069             Please provide:
1070             1. An LLM prompt to extract a smaller representation of
1071 what is relevant to the join task. The prompt should be a Jinja2
1072 template, referring to any fields in the input data as {{{{
1073 input.field_name }}}}. The prompt should instruct the LLM to return some
1074 **non-empty** string-valued output. The transformation should be
1075 tailored to the join task if possible, not just a generic summary of the
1076 data.
1077             2. A name for the new output key that will store the
1078 transformed data.
1079             3. An edited comparison prompt that leverages the new
1080 attribute created by the transformation. This prompt should be a Jinja2
1081 template, referring to any fields in the input data as {{{{
1082 left.field_name }}}}} and {{{{{ right.field_name }}}}}. The prompt should
1083 be the same as the current comparison prompt, but with a new instruction
1084 that leverages the new attribute created by the transformation (in
1085 addition to the other fields in the prompt). The prompt should instruct
1086 the LLM to return a boolean-valued output, like the current comparison
1087 prompt."""",
1088         }
1089     ]
1090
1091     response = self.llm_client.generate_rewrite(
1092         messages,
1093         "You are an AI expert in data analysis and decomposing
1094 complex data processing pipelines.",
1095         {
1096             "type": "object",
1097             "properties": {
1098                 "extraction_prompt": {"type": "string"},
1099                 "output_key": {"type": "string"},
1100                 "new_comparison_prompt": {"type": "string"},
1101             },
1102             "required": [
1103                 "extraction_prompt",
1104                 "output_key",
1105                 "new_comparison_prompt",
1106             ],
1107         },
1108     )

```

```

1109         result = json.loads(response.choices[0].message.content)
1110
1111     return (
1112         result["extraction_prompt"]
1113         .replace("left.", "input.")
1114         .replace("right.", "input."),
1115         result["output_key"],
1116         result["new_comparison_prompt"],
1117     )
1118
1119
1120     def _generate_blocking_keys_equijoin(
1121         self,
1122         left_data: list[dict[str, Any]],
1123         right_data: list[dict[str, Any]],
1124         sample_size: int = 5,
1125     ) -> tuple[list[str], list[str]]:
1126         # Sample data
1127         left_sample = random.sample(left_data, min(sample_size,
1128             len(left_data)))
1129         right_sample = random.sample(right_data, min(sample_size,
1130             len(right_data)))
1131
1132         # Prepare sample data for LLM
1133         left_keys = list(left_sample[0].keys())
1134         right_keys = list(right_sample[0].keys())
1135
1136         messages = [
1137             {
1138                 "role": "user",
1139                 "content": f"""Given the following sample data from two
1140 datasets, select appropriate blocking keys for an equijoin operation.
1141 The blocking process works as follows:
1142 1. We create embeddings for the selected keys from both
1143 datasets.
1144 2. We use cosine similarity between these embeddings to
1145 filter pairs for more detailed LLM comparison.
1146 3. Pairs with high similarity will be passed to the LLM
1147 for final comparison.
1148
1149             The blocking keys should have relatively short values
1150             and be useful for generating embeddings that capture the essence of
1151             potential matches.
1152
1153             Left dataset keys: {left_keys}
1154             Right dataset keys: {right_keys}
1155
1156             Sample from left dataset:
1157             {json.dumps(left_sample, indent=2)}
1158
1159             Sample from right dataset:
1160             {json.dumps(right_sample, indent=2)}
1161
1162             For context, here is the comparison prompt that will be
1163             used for the more detailed LLM comparison:
1164             {self.op_config.get('comparison_prompt', 'No comparison
1165             prompt provided.')}
1166
1167             Please select one or more keys from each dataset that
1168             would be suitable for blocking. The keys should contain information
1169             that's likely to be similar in matching records and align with the

```

```

1170     comparison prompt's focus."",
1171         }
1172     ]
1173
1174     response = self.llm_client.generate_rewrite(
1175         messages,
1176         "You are an expert in entity matching and database
operations.",
1177         {
1178             "type": "object",
1179             "properties": {
1180                 "left_blocking_keys": {
1181                     "type": "array",
1182                     "items": {"type": "string"},
1183                     "description": "List of selected blocking keys
from the left dataset",
1184                     },
1185                     "right_blocking_keys": {
1186                         "type": "array",
1187                         "items": {"type": "string"},
1188                         "description": "List of selected blocking keys
from the right dataset",
1189                         },
1190                         },
1191                         "required": ["left_blocking_keys",
1192 "right_blocking_keys"],
1193                         },
1194                         )
1195
1196
1197
1198
1199     result = json.loads(response.choices[0].message.content)
1200     left_blocking_keys = result["left_blocking_keys"]
1201     right_blocking_keys = result["right_blocking_keys"]
1202
1203     return left_blocking_keys, right_blocking_keys
1204
1205     def _compute_embeddings(
1206         self,
1207         input_data: list[dict[str, Any]],
1208         keys: list[str] | None = None,
1209         is_join: bool = True,
1210     ) -> tuple[list[list[float]], list[str], float]:
1211         if keys is None:
1212             keys = self.op_config.get("blocking_keys", [])
1213             if not keys:
1214                 prompt_template =
1215 self.op_config.get("comparison_prompt", "")
1216                 prompt_vars = extract_jinja_variables(prompt_template)
1217                 # Get rid of input, input1, input2
1218                 prompt_vars = [
1219                     var
1220                     for var in prompt_vars
1221                     if var not in ["input", "input1", "input2"]
1222                 ]
1223
1224                 # strip all things before . in the prompt_vars
1225                 keys += list(set([var.split(".")[-1] for var in
1226 prompt_vars]))
1227                 if not keys:
1228                     self.console.log(
1229                         "[yellow]Warning: No blocking keys found. Using all
keys for blocking.[/yellow]"
1230

```

```

1231             )
1232         keys = list(input_data[0].keys())
1233
1234     model_input_context_length = model_cost.get(
1235         self.op_config.get("embedding_model", "text-embedding-3-
1236 small"), {}
1237     ).get("max_input_tokens", 8192)
1238     texts = [
1239         " ".join(str(item[key])) for key in keys if key in item] [
1240             :model_input_context_length
1241         ]
1242         for item in input_data
1243     ]
1244
1245     embeddings = []
1246     total_cost = 0
1247     batch_size = 2000
1248     for i in range(0, len(texts), batch_size):
1249         batch = texts[i : i + batch_size]
1250         self.console.log(
1251             f"[cyan]Processing batch {i//batch_size + 1} of
{len(texts)//batch_size + 1}[/cyan]"
1252             )
1253         response = self.runner.api.gen_embedding(
1254             model=self.op_config.get("embedding_model", "text-
1255 embedding-3-small"),
1256             input=batch,
1257         )
1258         embeddings.extend([data["embedding"] for data in
1259             response["data"]])
1260         total_cost += completion_cost(response)
1261     embeddings = [data["embedding"] for data in response["data"]]
1262     cost = completion_cost(response)
1263     return embeddings, keys, cost
1264
1265
1266     def _calculate_cosine_similarities(
1267         self, embeddings: list[list[float]]
1268     ) -> list[tuple[int, int, float]]:
1269         embeddings_array = np.array(embeddings)
1270         norms = np.linalg.norm(embeddings_array, axis=1)
1271         dot_products = np.dot(embeddings_array, embeddings_array.T)
1272         similarities_matrix = dot_products / np.outer(norms, norms)
1273         i, j = np.triu_indices(len(embeddings), k=1)
1274         similarities = list(
1275             zip(i.tolist(), j.tolist(), similarities_matrix[i,
1276                 j].tolist())
1277         )
1278         return similarities
1279
1280     def _print_similarity_histogram(
1281         self,
1282         similarities: list[tuple[int, int, float]],
1283         comparison_results: list[tuple[int, int, bool]],
1284     ):
1285         flat_similarities = [sim[-1] for sim in similarities if sim[-1]
1286         != 1]
1287         hist, bin_edges = np.histogram(flat_similarities, bins=20)
1288         max_bar_width, max_count = 50, max(hist)
1289         normalized_hist = [int(count / max_count * max_bar_width) for
1290             count in hist]
1291

```

```

1292         # Create a dictionary to store true labels
1293         true_labels = {(i, j): is_match for i, j, is_match in
1294         comparison_results}
1295
1296         self.console.log("\n[bold]Embedding Cosine Similarity
1297 Distribution:[/bold]")
1298         for i, count in enumerate(normalized_hist):
1299             bar = "█" * count
1300             label = f"[bin_edges[{i}]:.2f]-[bin_edges[{i+1}]:.2f]"
1301
1302             # Count true matches and not matches in this bin
1303             true_matches = 0
1304             not_matches = 0
1305             labeled_count = 0
1306             for sim in similarities:
1307                 if bin_edges[i] <= sim[2] < bin_edges[i + 1]:
1308                     if (sim[0], sim[1]) in true_labels:
1309                         labeled_count += 1
1310                         if true_labels[(sim[0], sim[1])]:
1311                             true_matches += 1
1312                         else:
1313                             not_matches += 1
1314
1315             # Calculate percentages of labeled pairs
1316             if labeled_count > 0:
1317                 true_match_percent = (true_matches / labeled_count) *
1318                 100
1319                 not_match_percent = (not_matches / labeled_count) * 100
1320             else:
1321                 true_match_percent = 0
1322                 not_match_percent = 0
1323
1324             self.console.log(
1325                 f"{label}: {bar} "
1326                 f"(Labeled: {labeled_count}/{hist[i]}, [green]
1327 {true_match_percent:.1f}% match[/green], [red]{not_match_percent:.1f}%
1328 not match[/red])"
1329             )
1330             self.console.log("\n")
1331
1332     def _sample_pairs(
1333         self, similarities: list[tuple[int, int, float]]
1334     ) -> list[tuple[int, int]]:
1335         # Sort similarities in descending order
1336         sorted_similarities = sorted(similarities, key=lambda x: x[2],
1337         reverse=True)
1338
1339         # Calculate weights using exponential weighting with
1340         self.sampling_weight
1341         similarities_array = np.array([sim[2] for sim in
1342         sorted_similarities])
1343         weights = np.exp(self.sampling_weight * similarities_array)
1344         weights /= weights.sum() # Normalize weights to sum to 1
1345
1346         # Sample pairs based on the calculated weights
1347         sampled_indices = np.random.choice(
1348             len(sorted_similarities),
1349             size=min(self.sample_size, len(sorted_similarities)),
1350             replace=False,
1351             p=weights,
1352         )

```

```

1353
1354     sampled_pairs = [
1355         (sorted_similarities[i][0], sorted_similarities[i][1])
1356         for i in sampled_indices
1357     ]
1358     return sampled_pairs
1359
1360     def _calculate_cross_similarities(
1361         self, left_embeddings: list[list[float]], right_embeddings:
1362         list[list[float]]
1363         ) -> list[tuple[int, int, float]]:
1364             left_array = np.array(left_embeddings)
1365             right_array = np.array(right_embeddings)
1366             dot_product = np.dot(left_array, right_array.T)
1367             norm_left = np.linalg.norm(left_array, axis=1)
1368             norm_right = np.linalg.norm(right_array, axis=1)
1369             similarities = dot_product / np.outer(norm_left, norm_right)
1370             return [
1371                 (i, j, sim)
1372                 for i, row in enumerate(similarities)
1373                     for j, sim in enumerate(row)
1374             ]
1375
1376     def _perform_comparisons_resolve(
1377         self, input_data: list[dict[str, Any]], pairs: list[tuple[int,
1378         int]]
1379         ) -> tuple[list[tuple[int, int, bool]], float]:
1380             comparisons, total_cost = [], 0
1381             op = ResolveOperation(
1382                 self.runner,
1383                 self.op_config,
1384                 self.runner.default_model,
1385                 self.max_threads,
1386                 self.console,
1387                 self.status,
1388             )
1389             with ThreadPoolExecutor(max_workers=self.max_threads) as
1390             executor:
1391                 futures = [
1392                     executor.submit(
1393                         op.compare_pair,
1394                         self.op_config["comparison_prompt"],
1395                         self.op_config.get(
1396                             "comparison_model", self.config.get("model",
1397                             "gpt-4o-mini")
1398                         ),
1399                         input_data[i],
1400                         input_data[j],
1401                     )
1402                         for i, j in pairs
1403                     ]
1404             for future, (i, j) in zip(futures, pairs):
1405                 is_match, cost, _ = future.result()
1406                 comparisons.append((i, j, is_match))
1407                 total_cost += cost
1408
1409             self.console.log(
1410                 f"[bold]Cost of pairwise comparisons on the sample:
1411 ${total_cost:.4f}[/bold]"
1412             )
1413             return comparisons, total_cost

```

```

1414
1415     def _perform_comparisons_equijoin(
1416         self,
1417         left_data: list[dict[str, Any]],
1418         right_data: list[dict[str, Any]],
1419         pairs: list[tuple[int, int]],
1420     ) -> tuple[list[tuple[int, int, bool]], float]:
1421         comparisons, total_cost = [], 0
1422         op = EquijoinOperation(
1423             self.runner,
1424             self.op_config,
1425             self.runner.default_model,
1426             self.max_threads,
1427             self.console,
1428             self.status,
1429         )
1430         with ThreadPoolExecutor(max_workers=self.max_threads) as
1431             executor:
1432                 futures = [
1433                     executor.submit(
1434                         op.compare_pair,
1435                         self.op_config["comparison_prompt"],
1436                         self.op_config.get(
1437                             "comparison_model", self.config.get("model",
1438                             "gpt-4o-mini")
1439                         ),
1440                         left_data[i],
1441                         right_data[j] if right_data else left_data[j],
1442                     )
1443                     for i, j in pairs
1444                 ]
1445                 for future, (i, j) in zip(futures, pairs):
1446                     is_match, cost = future.result()
1447                     comparisons.append((i, j, is_match))
1448                     total_cost += cost
1449
1450                     self.console.log(
1451                         f"[bold]Cost of pairwise comparisons on the sample:
1452 ${total_cost:.4f}[/bold]"
1453                     )
1454             return comparisons, total_cost
1455
1456     def _find_optimal_threshold(
1457         self,
1458         comparisons: list[tuple[int, int, bool]],
1459         similarities: list[tuple[int, int, float]],
1460     ) -> tuple[float, float, float]:
1461         true_labels = np.array([comp[2] for comp in comparisons])
1462         sim_dict = {(i, j): sim for i, j, sim in similarities}
1463         sim_scores = np.array([sim_dict[(i, j)] for i, j, _ in
1464         comparisons])
1465
1466         thresholds = np.linspace(0, 1, 100)
1467         precisions, recalls = [], []
1468
1469         for threshold in thresholds:
1470             predictions = sim_scores >= threshold
1471             tp = np.sum(predictions & true_labels)
1472             fp = np.sum(predictions & ~true_labels)
1473             fn = np.sum(~predictions & true_labels)
1474

```

```

1475         precision = tp / (tp + fp) if (tp + fp) > 0 else 0
1476         recall = tp / (tp + fn) if (tp + fn) > 0 else 0
1477
1478         precisions.append(precision)
1479         recalls.append(recall)
1480
1481     valid_indices = [i for i, r in enumerate(recalls) if r >=
1482 self.target_recall]
1483     if not valid_indices:
1484         optimal_threshold = float(thresholds[np.argmax(recalls)])
1485     else:
1486         optimal_threshold = float(thresholds[max(valid_indices)])
1487
1488     # Improved selectivity estimation
1489     all_similarities = np.array([s[2] for s in similarities])
1490     sampled_similarities = sim_scores
1491
1492     # Calculate sampling probabilities
1493     sampling_probs = np.exp(self.sampling_weight *
1494 sampled_similarities)
1495     sampling_probs /= sampling_probs.sum()
1496
1497     # Estimate selectivity using importance sampling
1498     weights = 1 / (len(all_similarities) * sampling_probs)
1499     numerator = np.sum(weights * true_labels)
1500     denominator = np.sum(weights)
1501     selectivity_estimate = numerator / denominator
1502
1503     self.console.log(
1504         "[bold cyan]└ Estimated Self-Join Selectivity"
1505         "[/bold cyan]"
1506     )
1507     self.console.log(
1508         f"[bold cyan]└ [yellow]Target Recall:[/yellow]"
1509         f"{self.target_recall:.0%}"
1510     )
1511     self.console.log(
1512         f"[bold cyan]└ [yellow]Estimate:[/yellow]"
1513         f"{selectivity_estimate:.4f}"
1514     )
1515     self.console.log(
1516         "[bold"
1517         "cyan]└ [yellow]Threshold for blocking:"
1518         "[/bold cyan]"
1519     )
1520     self.console.log(
1521         f"[bold]Chosen similarity threshold for blocking:"
1522         f"{optimal_threshold:.4f}[/bold]"
1523     )
1524
1525     return round(optimal_threshold, 4), selectivity_estimate
1526
1527     def _generate_blocking_rules(
1528         self,
1529         blocking_keys: list[str],
1530         input_data: list[dict[str, Any]],
1531         comparisons: list[tuple[int, int, bool]],
1532     ) -> list[str]:
1533         # Sample 2 true and 2 false comparisons
1534         true_comparisons = [comp for comp in comparisons if comp[2]][:-2]
1535         false_comparisons = [comp for comp in comparisons if not

```

```

1536     comp[2]][:2]
1537         sample_datas = [
1538             (
1539                 {key: input_data[i][key] for key in blocking_keys},
1540                 {key: input_data[j][key] for key in blocking_keys},
1541                 is_match,
1542             )
1543             for i, j, is_match in true_comparisons + false_comparisons
1544         ]
1545
1546         messages = [
1547             {
1548                 "role": "user",
1549                 "content": f"""Given the following sample comparisons
1550 between entities, generate a single-line Python statement that acts as a
1551 blocking rule for entity resolution. This rule will be used in the form:
1552 `eval(blocking_rule, {{"input1": item1, "input2": item2}})`.
1553
1554 Sample comparisons (note: these are just a few examples and may not
1555 represent all possible cases):
1556             {json.dumps(sample_datas, indent=2)}
1557
1558             For context, here is the comparison prompt that will be used for the
1559 more expensive, detailed comparison:
1560             {self.op_config.get('comparison_prompt', 'No comparison prompt
1561 provided.')}
1562
1563             Please generate ONE one-line blocking rule that adheres to the
1564 following criteria:
1565             1. The rule should evaluate to True if the entities are possibly a
1566 match and require further comparison.
1567             2. The rule should evaluate to False ONLY if the entities are
1568 definitely not a match.
1569             3. The rule must be a single Python expression that can be evaluated
1570 using the eval() function.
1571             4. The rule should be much faster to evaluate than the full
1572 comparison prompt.
1573             5. The rule should capture the essence of the comparison prompt but
1574 in a simplified manner.
1575             6. The rule should be general enough to work well on the entire
1576 dataset, not just these specific examples.
1577             7. The rule should handle inconsistent casing by using string
1578 methods like .lower() when comparing string values.
1579             8. The rule should err on the side of inclusivity - it's better to
1580 have false positives than false negatives.
1581
1582             Example structure of a one-line blocking rule:
1583             "(condition1) or (condition2) or (condition3)"
1584
1585             Where conditions could be comparisons like:
1586             "input1['field'].lower() == input2['field'].lower()"
1587             "abs(len(input1['text']) - len(input2['text'])) <= 5"
1588             "any(word in input1['description'].lower() for word in
1589             input2['description'].lower().split())"
1590
1591             If there's no clear rule that can be generated based on the given
1592 information, return the string "True" to ensure all pairs are compared.
1593
1594             Remember, the primary goal of the blocking rule is to safely reduce
1595 the number of comparisons by quickly identifying pairs that are
1596 definitely not matches, while keeping all potential matches for further

```

```

1597     evaluation."""",
1598         }
1599     ]
1600
1601     for attempt in range(self.agent_max_retries): # Up to 3
attempts
1602         # Generate blocking rule using the LLM
1603         response = self.llm_client.generate_rewrite(
1604             messages,
1605             "You are an expert in entity resolution and Python
programming. Your task is to generate one efficient blocking rule based
on the given sample comparisons and data structure.",
1606             {
1607                 "type": "object",
1608                 "properties": {
1609                     "blocking_rule": {
1610                         "type": "string",
1611                         "description": "One-line Python statement
acting as a blocking rule",
1612                         }
1613                     },
1614                     "required": ["blocking_rule"],
1615                 },
1616             )
1617
1618         # Extract the blocking rule from the LLM response
1619         blocking_rule = response.choices[0].message.content
1620         blocking_rule =
1621         json.loads(blocking_rule).get("blocking_rule")
1622
1623         if blocking_rule:
1624             self.console.log("") # Print a newline
1625
1626             if blocking_rule.strip() == "True":
1627                 self.console.log(
1628                     "[yellow]No suitable blocking rule could be
found. Proceeding without a blocking rule.[/yellow]"
1629                 )
1630             return []
1631
1632             self.console.log(
1633                 f"[bold]Generated blocking rule (Attempt {attempt +
1634 1}):[/bold] {blocking_rule}"
1635                 )
1636
1637             # Test the blocking rule
1638             filtered_pairs = self._test_blocking_rule(
1639                 input_data, blocking_keys, blocking_rule,
comparisons
1640             )
1641
1642             if not filtered_pairs:
1643                 self.console.log(
1644                     "[green]Blocking rule looks good! No known
matches were filtered out.[/green]"
1645                 )
1646             return [blocking_rule]
1647             else:
1648                 feedback = f"The previous rule incorrectly filtered
out {len(filtered_pairs)} known matches. "
1649                 feedback += (

```

```

1658                     "Here are up to 3 examples of incorrectly
1659         filtered pairs:\n"
1660             )
1661             for i, j in filtered_pairs[:3]:
1662                 feedback += f"Item 1: {json.dumps({key:
1663 input_data[i][key] for key in blocking_keys})}\nItem 2:
1664 {json.dumps({key: input_data[j][key] for key in blocking_keys})}\n"
1665             feedback += "These pairs are known matches but
1666 were filtered out by the rule.\n"
1667             feedback += "Please generate a new rule that doesn't
1668 filter out these matches."
1669
1670             messages.append({"role": "assistant", "content":
1671 blocking_rule})
1672             messages.append({"role": "user", "content":
1673 feedback})
1674         else:
1675             self.console.log("[yellow]No blocking rule generated.
1676 [/yellow]")
1677             return []
1678
1679         self.console.log(
1680             f"[yellow]Failed to generate a suitable blocking rule after
1681 {self.agent_max_retries} attempts. Proceeding without a blocking rule.
1682 [/yellow]"
1683         )
1684         return []
1685
1686     def _test_blocking_rule(
1687         self,
1688         input_data: list[dict[str, Any]],
1689         blocking_keys: list[str],
1690         blocking_rule: str,
1691         comparisons: list[tuple[int, int, bool]],
1692     ) -> list[tuple[int, int]]:
1693         def apply_blocking_rule(item1, item2):
1694             try:
1695                 return eval(blocking_rule, {"input1": item1, "input2":
1696 item2})
1697             except Exception as e:
1698                 self.console.log(f"[red]Error applying blocking rule:
1699 {e}[/red]")
1700             return True # If there's an error, we default to
comparing the pair
1701
1702             filtered_pairs = []
1703
1704             for i, j, is_match in comparisons:
1705                 if is_match:
1706                     item1 = {
1707                         k: input_data[i][k] for k in blocking_keys if k in
1708 input_data[i]
1709                     }
1710                     item2 = {
1711                         k: input_data[j][k] for k in blocking_keys if k in
1712 input_data[j]
1713                     }
1714
1715                     if not apply_blocking_rule(item1, item2):
1716                         filtered_pairs.append((i, j))
1717
1718

```

```

1719         if filtered_pairs:
1720             self.console.log(
1721                 f"[yellow italic]LLM Correction: The blocking rule
1722                 incorrectly filtered out {len(filtered_pairs)} known positive matches.
1723                 [/yellow italic]"
1724             )
1725             for i, j in filtered_pairs[:5]: # Show up to 5 examples
1726                 self.console.log(
1727                     f" Incorrectly filtered pair 1: {json.dumps({key:
1728                         input_data[i][key] for key in blocking_keys})} and pair 2:
1729                         {json.dumps({key: input_data[j][key] for key in blocking_keys})}"
1730                     )
1731             if len(filtered_pairs) > 5:
1732                 self.console.log(
1733                     f" ... and {len(filtered_pairs) - 5} more incorrect
1734                 pairs."
1735             )
1736
1737             return filtered_pairs
1738
1739     def _generateContainmentRulesEquijoin(
1740         self,
1741         left_data: list[dict[str, Any]],
1742         right_data: list[dict[str, Any]],
1743     ) -> list[str]:
1744         # Get all available keys from the sample data
1745         left_keys = set(left_data[0].keys())
1746         right_keys = set(right_data[0].keys())
1747
1748         # Find the keys that are in the config's prompt
1749         try:
1750             left_prompt_keys = set(
1751                 self.op_config.get("comparison_prompt", "")
1752                 .split("{ left.")[1]
1753                 .split(" })")[0]
1754                 .split(".")
1755             )
1756         except Exception as e:
1757             self.console.log(f"[red]Error parsing comparison prompt: {e}
1758 [/red]")
1759             left_prompt_keys = left_keys
1760
1761         try:
1762             right_prompt_keys = set(
1763                 self.op_config.get("comparison_prompt", "")
1764                 .split("{ right.")[1]
1765                 .split(" })")[0]
1766                 .split(".")
1767             )
1768         except Exception as e:
1769             self.console.log(f"[red]Error parsing comparison prompt: {e}
1770 [/red]")
1771             right_prompt_keys = right_keys
1772
1773         # Sample a few records from each dataset
1774         sample_left = random.sample(left_data, min(3, len(left_data)))
1775         sample_right = random.sample(right_data, min(3,
1776             len(right_data)))
1777
1778         messages = [
1779             {

```

```
        "role": "system",
        "content": "You are an AI assistant tasked with
generating containment-based blocking rules for an equijoin operation.",
    },
    {
        "role": "user",
        "content": f"""Generate multiple one-line Python
statements that act as containment-based blocking rules for equijoin.
These rules will be used in the form: `eval(blocking_rule, {"left": item1, "right": item2})`.

Available keys in left dataset: {', '.join(left_keys)}
Available keys in right dataset: {', '.join(right_keys)}

Sample data from left dataset:
{json.dumps(sample_left, indent=2)}

Sample data from right dataset:
{json.dumps(sample_right, indent=2)}

Comparison prompt used for detailed comparison:
{self.op_config.get('comparison_prompt', 'No comparison prompt
provided.')}

Please generate multiple one-line blocking rules that adhere to the
following criteria:
1. The rules should focus on containment relationships between fields in
the left and right datasets. Containment can mean that the left field
contains all the words in the right field, or the right field contains
all the words in the left field.
2. Each rule should evaluate to True if there's a potential match based
on containment, False otherwise.
3. Rules must be single Python expressions that can be evaluated using
the eval() function.
4. Rules should handle inconsistent casing by using string methods like
.lower() when comparing string values.
5. Consider the length of the fields when generating rules: for example,
if the left field is much longer than the right field, it's more likely
to contain all the words in the right field.

Example structures of containment-based blocking rules:
"all(word in left['{{left_key}}'].lower() for word in
right['{{right_key}}'].lower().split())"
"any(word in right['{{right_key}}'].lower().split() for word in
left['{{left_key}}'].lower().split())"

Please provide 3-5 different containment-based blocking rules, based on
the keys and sample data provided. Prioritize rules with the following
keys: {', '.join(left_prompt_keys)} and {',
'.join(right_prompt_keys)}."""
    ],
}

response = self.llm_client.generate_rewrite(
    messages,
    "You are an expert in data matching and Python
programming.",
{
    "type": "object",
    "properties": {
        "containment_rules": {

```

```

        "type": "array",
        "items": {"type": "string"},
        "description": "List of containment-based
blocking rules as Python expressions",
    }
},
"required": ["containment_rules"],
),
)

containment_rules = response.choices[0].message.content
containment_rules =
json.loads(containment_rules).get("containment_rules")
return containment_rules

def _generate_blocking_rules_equijoin(
    self,
    left_keys: list[str],
    right_keys: list[str],
    left_data: list[dict[str, Any]],
    right_data: list[dict[str, Any]],
    comparisons: list[tuple[int, int, bool]],
) -> list[str]:
    if not left_keys or not right_keys:
        left_keys = list(left_data[0].keys())
        right_keys = list(right_data[0].keys())

    # Sample 2 true and 2 false comparisons
    true_comparisons = [comp for comp in comparisons if comp[2]][:2]
    false_comparisons = [comp for comp in comparisons if not
comp[2]][:2]
    sample_datas = [
        (
            {key: left_data[i][key] for key in left_keys if key in
left_data[i]},
            {key: right_data[j][key] for key in right_keys if key in
right_data[j]},
            is_match,
        )
        for i, j, is_match in true_comparisons + false_comparisons
    ]

    messages = [
        {
            "role": "user",
            "content": f"""Given the following sample comparisons
between entities, generate a single-line Python statement that acts as a
blocking rule for equijoin. This rule will be used in the form:
`eval(blocking_rule, {'left': item1, 'right': item2})`.

Sample comparisons (note: these are just a few examples and may not
represent all possible cases):
{json.dumps(sample_datas, indent=2)}
"""
        }
    ]
    for message in messages:
        response = self._client.chat.completions.create(
            model="gpt-4",
            messages=message["content"],
            temperature=0.5,
            max_tokens=1000,
            n=1,
            stop=None,
            timeout=None,
        )
        message["completion"] = response.choices[0].text
    return messages
}

For context, here is the comparison prompt that will be used for the
more expensive, detailed comparison:
{self.op_config.get('comparison_prompt', 'No comparison prompt
provided.')}

Please generate ONE one-line blocking rule that adheres to the
following criteria:
```

1. The rule should evaluate to True if the entities are possibly a match and require further comparison.
2. The rule should evaluate to False ONLY if the entities are definitely not a match.
3. The rule must be a single Python expression that can be evaluated using the eval() function.
4. The rule should be much faster to evaluate than the full comparison prompt.
5. The rule should capture the essence of the comparison prompt but in a simplified manner.
6. The rule should be general enough to work well on the entire dataset, not just these specific examples.
7. The rule should handle inconsistent casing by using string methods like .lower() when comparing string values.
8. The rule should err on the side of inclusivity - it's better to have false positives than false negatives.

Example structure of a one-line blocking rule:
 "(condition1) or (condition2) or (condition3)"

Where conditions could be comparisons like:

```
"left['{left_keys[0]}'].lower() == right['{right_keys[0]}'].lower()"  

"abs(len(left['{left_keys[0]}']) - len(right['{right_keys[0]}'])) <= 5"  

"any(word in left['{left_keys[0]}'].lower() for word in  

right['{right_keys[0]}'].lower().split())"
```

If there's no clear rule that can be generated based on the given information, return the string "True" to ensure all pairs are compared.

Remember, the primary goal of the blocking rule is to safely reduce the number of comparisons by quickly identifying pairs that are definitely not matches, while keeping all potential matches for further evaluation.""" ,

```
    }  

]  
  

for attempt in range(self.agent_max_retries):  

    response = self.llm_client.generate_rewrite(  

        messages,  

        "You are an expert in entity resolution and Python  

        programming. Your task is to generate one efficient blocking rule based  

        on the given sample comparisons and data structure.",  

        {  

            "type": "object",  

            "properties": {  

                "blocking_rule": {  

                    "type": "string",  

                    "description": "One-line Python statement  

acting as a blocking rule",  

                }  

            },  

            "required": ["blocking_rule"],  

        },  

    )  
  

    blocking_rule = response.choices[0].message.content  

    blocking_rule =  

    json.loads(blocking_rule).get("blocking_rule")  
  

    if blocking_rule:
```

```

        self.console.log("")

        if blocking_rule.strip() == "True":
            self.console.log(
                "[yellow]No suitable blocking rule could be
found. Proceeding without a blocking rule.[/yellow]")
        )
        return []

    self.console.log(
        f"[bold]Generated blocking rule (Attempt {attempt +
1}):[/bold] {blocking_rule}"
    )

    # Test the blocking rule
    filtered_pairs = self._test_blocking_rule_equijoin(
        left_data,
        right_data,
        left_keys,
        right_keys,
        blocking_rule,
        comparisons,
    )

    if not filtered_pairs:
        self.console.log(
            "[green]Blocking rule looks good! No known
matches were filtered out.[/green]"
        )
        return [blocking_rule]
    else:
        feedback = f"The previous rule incorrectly filtered
out {len(filtered_pairs)} known matches."
        feedback += (
            "Here are up to 3 examples of incorrectly
filtered pairs:\n"
        )
        for i, j in filtered_pairs[:3]:
            feedback += f"Left: {json.dumps({key:
left_data[i][key] for key in left_keys})}\n"
            feedback += f"Right: {json.dumps({key:
right_data[j][key] for key in right_keys})}\n"
            feedback += "These pairs are known matches but
were filtered out by the rule.\n"
        feedback += "Please generate a new rule that doesn't
filter out these matches."

        messages.append({"role": "assistant", "content": blocking_rule})
        messages.append({"role": "user", "content": feedback})
    else:
        self.console.log("[yellow]No blocking rule generated.
[/yellow]")
        return []

    self.console.log(
        f"[yellow]Failed to generate a suitable blocking rule after
{self.agent_max_retries} attempts. Proceeding without a blocking rule.
[/yellow]"
    )

```

```

        return []

def _test_blocking_rule_equijoin(
    self,
    left_data: list[dict[str, Any]],
    right_data: list[dict[str, Any]],
    left_keys: list[str],
    right_keys: list[str],
    blocking_rule: str,
    comparisons: list[tuple[int, int, bool]],
) -> list[tuple[int, int]]:
    def apply_blocking_rule(left, right):
        try:
            return eval(blocking_rule, {"left": left, "right": right})
        except Exception as e:
            self.console.log(f"[red]Error applying blocking rule: {e}[/red]")
        return True # If there's an error, we default to comparing the pair

    filtered_pairs = []

    for i, j, is_match in comparisons:
        if is_match:
            left = left_data[i]
            right = right_data[j]
            if not apply_blocking_rule(left, right):
                filtered_pairs.append((i, j))

    if filtered_pairs:
        self.console.log(
            f"[yellow italic]LLM Correction: The blocking rule incorrectly filtered out {len(filtered_pairs)} known positive matches. [/yellow italic]"
        )
        for i, j in filtered_pairs[:5]: # Show up to 5 examples
            left_dict = {key: left_data[i][key] for key in left_keys}
            right_dict = {key: right_data[j][key] for key in right_keys}
            self.console.log(
                f"  Incorrectly filtered pair - Left: {json.dumps(left_dict)}  Right: {json.dumps(right_dict)}"
            )
        if len(filtered_pairs) > 5:
            self.console.log(
                f"  ... and {len(filtered_pairs) - 5} more incorrect pairs."
            )

    return filtered_pairs

def _verify_blocking_rule_equijoin(
    self,
    left_data: list[dict[str, Any]],
    right_data: list[dict[str, Any]],
    blocking_rule: str,
    left_keys: list[str],
    right_keys: list[str],
    comparison_results: list[tuple[int, int, bool]],
)

```

```

) -> tuple[list[tuple[int, int]], float]:
def apply_blocking_rule(left, right):
    try:
        return eval(blocking_rule, {"left": left, "right": right})
    except Exception as e:
        self.console.log(f"[red]Error applying blocking rule: {e}[/red]")
    return True # If there's an error, we default to comparing the pair

    false_negatives = []
    total_pairs = 0
    blocked_pairs = 0

    for i, j, is_match in comparison_results:
        total_pairs += 1
        left = left_data[i]
        right = right_data[j]
        if apply_blocking_rule(left, right):
            blocked_pairs += 1
            if is_match:
                false_negatives.append((i, j))

    rule_selectivity = blocked_pairs / total_pairs if total_pairs > 0 else 0

    return false_negatives, rule_selectivity

def _update_config_equijoin(
    self,
    threshold: float,
    left_keys: list[str],
    right_keys: list[str],
    blocking_rules: list[str],
) -> dict[str, Any]:
    optimized_config = self.op_config.copy()
    optimized_config["blocking_keys"] = {
        "left": left_keys,
        "right": right_keys,
    }
    optimized_config["blocking_threshold"] = threshold
    if blocking_rules:
        optimized_config["blocking_conditions"] = blocking_rules
    if "embedding_model" not in optimized_config:
        optimized_config["embedding_model"] = "text-embedding-3-small"
    return optimized_config

def _verify_blocking_rule(
    self,
    input_data: list[dict[str, Any]],
    blocking_rule: str,
    blocking_keys: list[str],
    comparison_results: list[tuple[int, int, bool]],
) -> tuple[list[tuple[int, int]], float]:
    def apply_blocking_rule(item1, item2):
        try:
            return eval(blocking_rule, {"input1": item1, "input2": item2})
        except Exception as e:

```

```

        self.console.log(f"[red]Error applying blocking rule:
{e}[/red]")
    return True # If there's an error, we default to
comparing the pair

    false_negatives = []
    total_pairs = 0
    blocked_pairs = 0

    for i, j, is_match in comparison_results:
        total_pairs += 1
        item1 = {k: input_data[i][k] for k in blocking_keys if k in
input_data[i]}
        item2 = {k: input_data[j][k] for k in blocking_keys if k in
input_data[j]}

        if apply_blocking_rule(item1, item2):
            blocked_pairs += 1
            if is_match:
                false_negatives.append((i, j))

    rule_selectivity = blocked_pairs / total_pairs if total_pairs >
0 else 0

    return false_negatives, rule_selectivity

def _update_config(
    self, threshold: float, blocking_keys: list[str],
blocking_rules: list[str]
) -> dict[str, Any]:
    optimized_config = self.op_config.copy()
    optimized_config["blocking_keys"] = blocking_keys
    optimized_config["blocking_threshold"] = threshold
    if blocking_rules:
        optimized_config["blocking_conditions"] = blocking_rules
    if "embedding_model" not in optimized_config:
        optimized_config["embedding_model"] = "text-embedding-3-
small"
    return optimized_config

```

`should_optimize(input_data)`

Determine if the given operation configuration should be optimized.

Source code in `docetl/optimizers/join_optimizer.py`

```
377     def should_optimize(self, input_data: list[dict[str, Any]]) ->
378         tuple[bool, str]:
379             """
380                 Determine if the given operation configuration should be optimized.
381             """
382             # If there are no blocking keys or embeddings, then we don't need to
383             # optimize
384             if not self.op_config.get("blocking_conditions") or not
385                 self.op_config.get(
386                     "blocking_threshold"
387                 ):
388                 return True, ""
389
390             # Check if the operation is marked as empty
391             elif self.op_config.get("empty", False):
392                 # Extract the map prompt from the intermediates
393                 map_prompt = self.op_config["_intermediates"]["map_prompt"]
394                 reduce_key = self.op_config["_intermediates"]["reduce_key"]
395
396                 if reduce_key is None:
397                     raise ValueError(
398                         "[yellow]Warning: No reduce key found in intermediates"
399                         for synthesized resolve operation.[/yellow]"
400                     )
401
402                 dedup = True
403                 explanation = "There is a reduce operation that does not follow a
404                 resolve operation. Consider adding a resolve operation to deduplicate the
405                 data."
406
407                 if map_prompt:
408                     # Analyze the map prompt
409                     analysis, explanation =
410                         self._analyze_map_prompt_categorization(
411                             map_prompt
412                         )
413
414                     if analysis:
415                         dedup = False
416                     else:
417                         self.console.log(
418                             "[yellow]No map prompt found in intermediates for
419                             analysis.[/yellow]"
420                         )
421
422                     # TODO: figure out why this would ever be the case
423                     if not map_prompt:
424                         map_prompt = "N/A"
425
426                     if dedup is False:
427                         dedup, explanation = self._determine_duplicate_keys(
428                             input_data, reduce_key, map_prompt
429                         )
430
431                     # Now do the last attempt of pairwise comparisons
432                     if dedup is False:
433                         # Sample up to 20 random pairs of keys for duplicate analysis
```

```
434         sampled_pairs = self._sample_random_pairs(input_data, 20)
435
436         # Use LLM to check for duplicates
437         duplicates_found, explanation =
438     self._check_duplicates_with_llm(
439             input_data, sampled_pairs, reduce_key, map_prompt
440         )
441
442         if duplicates_found:
443             dedup = True
444
445         return dedup, explanation
446
447     return False, ""
```

Python API

Operations

```
docetl.schemas.MapOp = map.MapOperation.schema module-attribute

docetl.schemas.ResolveOp = resolve.ResolveOperation.schema module-
attribute

docetl.schemas.ReduceOp = reduce.ReduceOperation.schema module-
attribute

docetl.schemas.ParallelMapOp = map.ParallelMapOperation.schema module-
attribute

docetl.schemas.FilterOp = filter.FilterOperation.schema module-
attribute

docetl.schemas.EquijoinOp = equijoin.EquijoinOperation.schema module-
attribute

docetl.schemas.SplitOp = split.SplitOperation.schema module-attribute

docetl.schemas.GatherOp = gather.GatherOperation.schema module-
attribute

docetl.schemas.UnnestOp = unnest.UnnestOperation.schema module-
attribute

docetl.schemas.SampleOp = sample.SampleOperation.schema module-
attribute
```

```
docetl.schemas.ClusterOp = cluster.ClusterOperation.schema module-attribute
```

Dataset and Pipeline

```
docetl.schemas.Dataset = dataset.Dataset.schema module-attribute
```

```
docetl.schemas.ParsingTool
```

Bases: `BaseModel`

Represents a parsing tool used for custom data parsing in the pipeline.

Attributes:

Name	Type	Description
<code>name</code>	<code>str</code>	The name of the parsing tool. This should be unique within the pipeline configuration.
<code>function_code</code>	<code>str</code>	The Python code defining the parsing function. This code will be executed to parse the input data according to the specified logic. It should return a list of strings, where each string is its own document.



Example

```
parsing_tools:
- name: ocr_parser
  function_code: |
    import pytesseract
    from pdf2image import convert_from_path
    def ocr_parser(filename: str) -> list[str]:
        images = convert_from_path(filename)
        text = ""
        for image in images:
            text += pytesseract.image_to_string(image)
    return [text]
```

Source code in `docetl/base_schemas.py`

```
20 class ParsingTool(BaseModel):
21     """
22         Represents a parsing tool used for custom data parsing in the
23         pipeline.
24
25     Attributes:
26         name (str): The name of the parsing tool. This should be unique
27         within the pipeline configuration.
28         function_code (str): The Python code defining the parsing
29         function. This code will be executed
30             to parse the input data according to the
31         specified logic. It should return a list of strings, where each string is
32         its own document.
33
34     Example:
35         ```yaml
36             parsing_tools:
37                 - name: ocr_parser
38                     function_code: |
39                         import pytesseract
40                         from pdf2image import convert_from_path
41                         def ocr_parser(filename: str) -> list[str]:
42                             images = convert_from_path(filename)
43                             text = ""
44                             for image in images:
45                                 text += pytesseract.image_to_string(image)
46                         return [text]
47
48             ...
49
50         """
51
52     name: str
53     function_code: str
```

`docetl.schemas.PipelineStep`

Bases: `BaseModel`

Represents a step in the pipeline.

Attributes:

Name	Type	Description
<code>name</code>	<code>str</code>	The name of the step.

Name	Type	Description
operations	list[dict[str, Any] str]	A list of operations to be applied in this step. Each operation can be either a string (the name of the operation) or a dictionary (for more complex configurations).
input	str None	The input for this step. It can be either the name of a dataset or the name of a previous step. If not provided, the step will use the output of the previous step as its input.

Example

```
# Simple step with a single operation
process_step = PipelineStep(
    name="process_step",
    input="my_dataset",
    operations=["process"]
)

# Step with multiple operations
summarize_step = PipelineStep(
    name="summarize_step",
    input="process_step",
    operations=["summarize"]
)

# Step with a more complex operation configuration
custom_step = PipelineStep(
    name="custom_step",
    input="previous_step",
    operations=[
        {
            "custom_operation": {
                "model": "gpt-4",
                "prompt": "Perform a custom analysis on the following text:"
            }
        }
    ]
)
```

These examples show different ways to configure pipeline steps, from simple single-operation steps to more complex configurations with custom parameters.

Source code in `docetl/base_schemas.py`

```
49 class PipelineStep(BaseModel):
50     """
51     Represents a step in the pipeline.
52
53     Attributes:
54         name (str): The name of the step.
55         operations (list[dict[str, Any] | str]): A list of operations to
56         be applied in this step.
57             Each operation can be either a string (the name of the
58             operation) or a dictionary
59                 (for more complex configurations).
60             input (str | None): The input for this step. It can be either the
61             name of a dataset
62                 or the name of a previous step. If not provided, the step will
63             use the output
64                 of the previous step as its input.
65
66     Example:
67         ```python
68         # Simple step with a single operation
69         process_step = PipelineStep(
70             name="process_step",
71             input="my_dataset",
72             operations=["process"]
73         )
74
75         # Step with multiple operations
76         summarize_step = PipelineStep(
77             name="summarize_step",
78             input="process_step",
79             operations=["summarize"]
80         )
81
82         # Step with a more complex operation configuration
83         custom_step = PipelineStep(
84             name="custom_step",
85             input="previous_step",
86             operations=[
87                 {
88                     "custom_operation": {
89                         "model": "gpt-4",
90                         "prompt": "Perform a custom analysis on the
91                     following text:"
92                         }
93                 }
94             ]
95         )
96         ...
97
98     These examples show different ways to configure pipeline steps, from
99     simple
    single-operation steps to more complex configurations with custom
parameters.
    """
name: str
```

```
operations: list[dict[str, Any] | str]
input: str | None = None
```

docetl.schemas.PipelineOutput

Bases: `BaseModel`

Represents the output configuration for a pipeline.

Attributes:

Name	Type	Description
<code>type</code>	<code>str</code>	The type of output. This could be 'file', 'database', etc.
<code>path</code>	<code>str</code>	The path where the output will be stored. This could be a file path, database connection string, etc., depending on the type.
<code>intermediate_dir</code>	<code>str None</code>	The directory to store intermediate results, if applicable. Defaults to None.



Example

```
output = PipelineOutput(
    type="file",
    path="/path/to/output.json",
    intermediate_dir="/path/to/intermediate/results"
)
```

Source code in `docetl/base_schemas.py`

```

102     class PipelineOutput(BaseModel):
103         """
104             Represents the output configuration for a pipeline.
105
106             Attributes:
107                 type (str): The type of output. This could be 'file', 'database',
108                 etc.
109                 path (str): The path where the output will be stored. This could
110                 be a file path,
111                         database connection string, etc., depending on the
112                 type.
113                 intermediate_dir (str | None): The directory to store
114                 intermediate results,
115                         if applicable. Defaults to
116                 None.
117
118             Example:
119                 ```python
120                     output = PipelineOutput(
121                         type="file",
122                         path="/path/to/output.json",
123                         intermediate_dir="/path/to/intermediate/results"
124                     )
125                 ```
126
127
128             type: str
129             path: str
130             intermediate_dir: str | None = None

```

`docetl.api.Pipeline`

Represents a complete document processing pipeline.

Attributes:

Name	Type	Description
<code>name</code>	<code>str</code>	The name of the pipeline.
<code>datasets</code>	<code>dict[str, Dataset]</code>	A dictionary of datasets used in the pipeline, where keys are dataset names and values are <code>Dataset</code> objects.
<code>operations</code>	<code>list[OpType]</code>	A list of operations to be performed in the pipeline.

Name	Type	Description
steps	list[PipelineStep]	A list of steps that make up the pipeline.
output	PipelineOutput	The output configuration for the pipeline.
parsing_tools	list[ParsingTool]	A list of parsing tools used in the pipeline. Defaults to an empty list.
default_model	str None	The default language model to use for operations that require one. Defaults to None.



Example

```
def custom_parser(text: str) -> list[str]:
    # this will convert the text in the column to uppercase
    # You should return a list of strings, where each string is a separate
    document
    return [text.upper()]

pipeline = Pipeline(
    name="document_processing_pipeline",
    datasets={
        "input_data": Dataset(type="file", path="/path/to/input.json", parsing=[
            {"name": "custom_parser", "input_key": "content", "output_key": "uppercase_content"}]),
    },
    parsing_tools=[custom_parser],
    operations=[
        MapOp(
            name="process",
            type="map",
            prompt="Determine what type of document this is: {{ input.uppercase_content }}",
            output={"schema": {"document_type": "string"}}
        ),
        ReduceOp(
            name="summarize",
            type="reduce",
            reduce_key="document_type",
            prompt="Summarize the processed contents: {% for item in inputs %} {{ item.uppercase_content }} {% endfor %}",
            output={"schema": {"summary": "string"}}
        )
    ],
    steps=[
        PipelineStep(name="process_step", input="input_data", operations=["process"]),
        PipelineStep(name="summarize_step", input="process_step", operations=["summarize"])
    ],
    output=PipelineOutput(type="file", path="/path/to/output.json"),
    default_model="gpt-4o-mini"
)
```

This example shows a complete pipeline configuration with datasets, operations, steps, and output settings.

Source code in docetl/api.py

```
80  class Pipeline:
81      """
82          Represents a complete document processing pipeline.
83
84      Attributes:
85          name (str): The name of the pipeline.
86          datasets (dict[str, Dataset]): A dictionary of datasets used in
87          the pipeline,
88                                     where keys are dataset names and
89          values are Dataset objects.
90          operations (list[OpType]): A list of operations to be performed
91          in the pipeline.
92          steps (list[PipelineStep]): A list of steps that make up the
93          pipeline.
94          output (PipelineOutput): The output configuration for the
95          pipeline.
96          parsing_tools (list[ParsingTool]): A list of parsing tools used
97          in the pipeline.
98                                     Defaults to an empty list.
99          default_model (str | None): The default language model to use for
100         operations
101                                     that require one. Defaults to
102         None.
103
104     Example:
105         ```python
106             def custom_parser(text: str) -> list[str]:
107                 # this will convert the text in the column to uppercase
108                 # You should return a list of strings, where each string is a
109                 separate document
110                 return [text.upper()]
111
112             pipeline = Pipeline(
113                 name="document_processing_pipeline",
114                 datasets={
115                     "input_data": Dataset(type="file",
116 path="/path/to/input.json", parsing=[{"name": "custom_parser",
117 "input_key": "content", "output_key": "uppercase_content"}]),
118                     },
119                     parsing_tools=[custom_parser],
120                     operations=[
121                         MapOp(
122                             name="process",
123                             type="map",
124                             prompt="Determine what type of document this is: {{
125 input.uppercase_content }}",
126                             output={"schema": {"document_type": "string"}}
127                         ),
128                         ReduceOp(
129                             name="summarize",
130                             type="reduce",
131                             reduce_key="document_type",
132                             prompt="Summarize the processed contents: {% for item
133 in inputs %}{{ item.uppercase_content }} {% endfor %}",
134                             output={"schema": {"summary": "string"}}
135                         )
136                     ],
137                 )]
```

```
137         steps=[  
138             PipelineStep(name="process_step", input="input_data",  
139             operations=["process"]),  
140             PipelineStep(name="summarize_step", input="process_step",  
141             operations=["summarize"])  
142         ],  
143         output=PipelineOutput(type="file",  
144             path="/path/to/output.json"),  
145             default_model="gpt-4o-mini"  
146         )  
147     ...  
148  
149     This example shows a complete pipeline configuration with datasets,  
150     operations,  
151     steps, and output settings.  
152     """  
153  
154     def __init__(  
155         self,  
156         name: str,  
157         datasets: dict[str, Dataset],  
158         operations: list[OpType],  
159         steps: list[PipelineStep],  
160         output: PipelineOutput,  
161         parsing_tools: list[ParsingTool | Callable] = [],  
162         default_model: str | None = None,  
163         rate_limits: dict[str, int] | None = None,  
164         optimizer_config: dict[str, Any] = {},  
165         **kwargs,  
166     ):  
167         self.name = name  
168         self.datasets = datasets  
169         self.operations = operations  
170         self.steps = steps  
171         self.output = output  
172         self.parsing_tools = [  
173             (  
174                 tool  
175                 if isinstance(tool, ParsingTool)  
176                 else ParsingTool(  
177                     name=tool.__name__,  
178                     function_code=inspect.getsource(tool)  
179                     )  
180                 )  
181                 for tool in parsing_tools  
182             ]  
183         self.default_model = default_model  
184         self.rate_limits = rate_limits  
185         self.optimizer_config = optimizer_config  
186  
187         # Add other kwargs to self.other_config  
188         self.other_config = kwargs  
189  
190         self._load_env()  
191  
192     def _load_env(self):  
193         import os  
194  
195         from dotenv import load_dotenv  
196  
197         # Get the current working directory
```

```
198     cwd = os.getcwd()
199
200     # Load .env file from the current working directory if it exists
201     env_file = os.path.join(cwd, ".env")
202     if os.path.exists(env_file):
203         load_dotenv(env_file)
204
205     def optimize(
206         self,
207         max_threads: int | None = None,
208         resume: bool = False,
209         save_path: str | None = None,
210     ) -> "Pipeline":
211         """
212             Optimize the pipeline using the Optimizer.
213
214             Args:
215                 max_threads (int | None): Maximum number of threads to use
216                 for optimization.
217                 model (str): The model to use for optimization. Defaults to
218                 "gpt-4o".
219                 resume (bool): Whether to resume optimization from a previous
220                 state. Defaults to False.
221                 timeout (int): Timeout for optimization in seconds. Defaults
222                 to 60.
223
224             Returns:
225                 Pipeline: An optimized version of the pipeline.
226             """
227         config = self._to_dict()
228         runner = DSLRunner(
229             config,
230             base_name=os.path.join(os.getcwd(), self.name),
231             yaml_file_suffix=self.name,
232             max_threads=max_threads,
233         )
234         optimized_config, _ = runner.optimize(
235             resume=resume,
236             return_pipeline=False,
237             save_path=save_path,
238         )
239
240         updated_pipeline = Pipeline(
241             name=self.name,
242             datasets=self.datasets,
243             operations=self.operations,
244             steps=self.steps,
245             output=self.output,
246             default_model=self.default_model,
247             parsing_tools=self.parsing_tools,
248             optimizer_config=self.optimizer_config,
249         )
250         updated_pipeline._update_from_dict(optimized_config)
251         return updated_pipeline
252
253     def run(self, max_threads: int | None = None) -> float:
254         """
255             Run the pipeline using the DSLRunner.
256
257             Args:
258                 max_threads (int | None): Maximum number of threads to use
```

```
259     for execution.
260
261     Returns:
262         float: The total cost of running the pipeline.
263         """
264     config = self._to_dict()
265     runner = DSLRunner(
266         config,
267         base_name=os.path.join(os.getcwd(), self.name),
268         yaml_file_suffix=self.name,
269         max_threads=max_threads,
270     )
271     result = runner.load_run_save()
272     return result
273
274     def to_yaml(self, path: str) -> None:
275         """
276             Convert the Pipeline object to a YAML string and save it to a
277             file.
278
279             Args:
280                 path (str): Path to save the YAML file.
281
282             Returns:
283                 None
284             """
285     config = self._to_dict()
286     with open(path, "w") as f:
287         yaml.safe_dump(config, f)
288
289     print(f"[green]Pipeline saved to {path}[/green]")
290
291     def _to_dict(self) -> dict[str, Any]:
292         """
293             Convert the Pipeline object to a dictionary representation.
294
295             Returns:
296                 dict[str, Any]: Dictionary representation of the Pipeline.
297             """
298     d = {
299         "datasets": [
300             name: dataset.dict() for name, dataset in
301             self.datasets.items()
302         ],
303         "operations": [
304             {k: v for k, v in op.dict().items() if v is not None}
305             for op in self.operations
306         ],
307         "pipeline": {
308             "steps": [
309                 {k: v for k, v in step.dict().items() if v is not
310                 None}
311                 for step in self.steps
312             ],
313             "output": self.output.dict(),
314         },
315         "default_model": self.default_model,
316         "parsing_tools": (
317             [tool.dict() for tool in self.parsing_tools]
318             if self.parsing_tools
319             else None
320         )
321     }
322
323     return d
```

```

320         ),
321         "optimizer_config": self.optimizer_config,
322         **self.other_config,
323     }
324     if self.rate_limits:
325         d["rate_limits"] = self.rate_limits
326     return d
327
328     def _update_from_dict(self, config: dict[str, Any]):
329         """
330             Update the Pipeline object from a dictionary representation.
331
332             Args:
333                 config (dict[str, Any]): Dictionary representation of the
334             Pipeline.
335             """
336         self.datasets = {
337             name: Dataset(
338                 type=dataset["type"],
339                 source=dataset["source"],
340                 path=dataset["path"],
341                 parsing=dataset.get("parsing"),
342             )
343             for name, dataset in config["datasets"].items()
344         }
345         self.operations = []
346         for op in config["operations"]:
347             op_type = op.pop("type")
348             if op_type == "map":
349                 self.operations.append(MapOp(**op, type=op_type))
350             elif op_type == "resolve":
351                 self.operations.append(ResolveOp(**op, type=op_type))
352             elif op_type == "reduce":
353                 self.operations.append(ReduceOp(**op, type=op_type))
354             elif op_type == "parallel_map":
355                 self.operations.append(ParallelMapOp(**op, type=op_type))
356             elif op_type == "filter":
357                 self.operations.append(FilterOp(**op, type=op_type))
358             elif op_type == "equijoin":
359                 self.operations.append(EquijoinOp(**op, type=op_type))
360             elif op_type == "split":
361                 self.operations.append(SplitOp(**op, type=op_type))
362             elif op_type == "gather":
363                 self.operations.append(GatherOp(**op, type=op_type))
364             elif op_type == "unnest":
365                 self.operations.append(UnnestOp(**op, type=op_type))
366             elif op_type == "cluster":
367                 self.operations.append(ClusterOp(**op, type=op_type))
368             elif op_type == "sample":
369                 self.operations.append(SampleOp(**op, type=op_type))
370             self.steps = [PipelineStep(**step) for step in config["pipeline"]
371 ["steps"]]
372             self.output = PipelineOutput(**config["pipeline"]["output"])
373             self.default_model = config.get("default_model")
374             self.parsing_tools = (
375                 [ParsingTool(**tool) for tool in config.get("parsing_tools",
376                 [])]
377                 if config.get("parsing_tools")
378                 else []
379             )

```

```
optimize(max_threads=None, resume=False, save_path=None)
```

Optimize the pipeline using the Optimizer.

Parameters:

Name	Type	Description	Default
max_threads	int None	Maximum number of threads to use for optimization.	None
model	str	The model to use for optimization. Defaults to "gpt-4o".	<i>required</i>
resume	bool	Whether to resume optimization from a previous state. Defaults to False.	False
timeout	int	Timeout for optimization in seconds. Defaults to 60.	<i>required</i>

Returns:

Name	Type	Description
Pipeline	Pipeline	An optimized version of the pipeline.

Source code in `docetl/api.py`

```

187     def optimize(
188         self,
189         max_threads: int | None = None,
190         resume: bool = False,
191         save_path: str | None = None,
192     ) -> "Pipeline":
193         """
194             Optimize the pipeline using the Optimizer.
195
196             Args:
197                 max_threads (int | None): Maximum number of threads to use for
198                 optimization.
199                 model (str): The model to use for optimization. Defaults to "gpt-
200                 4o".
201                 resume (bool): Whether to resume optimization from a previous
202                 state. Defaults to False.
203                 timeout (int): Timeout for optimization in seconds. Defaults to
204                 60.
205
206             Returns:
207                 Pipeline: An optimized version of the pipeline.
208             """
209         config = self._to_dict()
210         runner = DSLRunner(
211             config,
212             base_name=os.path.join(os.getcwd(), self.name),
213             yaml_file_suffix=self.name,
214             max_threads=max_threads,
215         )
216         optimized_config, _ = runner.optimize(
217             resume=resume,
218             return_pipeline=False,
219             save_path=save_path,
220         )
221
222         updated_pipeline = Pipeline(
223             name=self.name,
224             datasets=self.datasets,
225             operations=self.operations,
226             steps=self.steps,
227             output=self.output,
228             default_model=self.default_model,
229             parsing_tools=self.parsing_tools,
             optimizer_config=self.optimizer_config,
         )
         updated_pipeline._update_from_dict(optimized_config)
         return updated_pipeline

```

`run(max_threads=None)`

Run the pipeline using the DSLRunner.

Parameters:

Name	Type	Description	Default
max_threads	int None	Maximum number of threads to use for execution.	None

Returns:

Name	Type	Description
float	float	The total cost of running the pipeline.

Source code in [docetl/api.py](#)

```

231 def run(self, max_threads: int | None = None) -> float:
232     """
233         Run the pipeline using the DSLRunner.
234
235     Args:
236         max_threads (int | None): Maximum number of threads to use for
237             execution.
238
239     Returns:
240         float: The total cost of running the pipeline.
241     """
242     config = self._to_dict()
243     runner = DSLRunner(
244         config,
245         base_name=os.path.join(os.getcwd(), self.name),
246         yaml_file_suffix=self.name,
247         max_threads=max_threads,
248     )
249     result = runner.load_run_save()
250     return result

```

to_yaml(path)

Convert the Pipeline object to a YAML string and save it to a file.

Parameters:

Name	Type	Description	Default
path	str	Path to save the YAML file.	<i>required</i>

Returns:

Type	Description
None	None

Source code in `docetl/api.py`

```
251 def to_yaml(self, path: str) -> None:
252     """
253     Convert the Pipeline object to a YAML string and save it to a file.
254
255     Args:
256         path (str): Path to save the YAML file.
257
258     Returns:
259         None
260     """
261     config = self._to_dict()
262     with open(path, "w") as f:
263         yaml.safe_dump(config, f)
264
265     print(f"[green]Pipeline saved to {path}[/green]")
```