

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
508             for future in futures:
509                 result = future.result()
510                 all_results.append(result)
511
512             return all_results
513
514         else:
515             item, output = items_and_outputs[0]
516             return [(_process_map_item(item, output),)]
517
518     return all_results
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
999

```

```

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
513     fold_batch_size = self.config["fold_batch_size"]
514     merge_batch_size = self.config["merge_batch_size"]
515     total_cost = 0
516     prompts = []
517
518     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":t["children"]})
224
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"[\dots {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"[\dots {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
<code>input_data</code>	<code>list[dict]</code>	A list of dictionaries to process.	<code>required</code>

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"}, 5},
128     #   {"id": 2, "details": {"color": "blue", "size": "medium", "stock": 3}, "color": "blue", "size": "medium"} 3}
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,
148                                         dict)):
149             raise TypeError(f"Value of unnest key '{key}' is not
150 iterable")
151
152         if level > 0 and not isinstance(item[key], (list, tuple, set,
153                                         dict)):
154             return [item]
155
156         if level >= depth:
157             return [item]
158
159         if isinstance(item[key], dict):
160             expand_fields = self.config.get("expand_fields")
161             if expand_fields is None:
162                 expand_fields = item[key].keys()
163             new_item = copy.deepcopy(item)
164             for field in expand_fields:
165                 if field in new_item[key]:
166                     new_item[field] = new_item[key][field]
167                 else:
168                     new_item[field] = None
169             return [new_item]
170         else:
171             nested_results = []
172             for value in item[key]:
173                 new_item = copy.deepcopy(item)
174                 new_item[key] = value
175                 if recursive and isinstance(value, (list, tuple, set,
176                                         dict)):
177                     nested_results.extend(
178                         unnest_recursive(new_item, key, level + 1)
179                     )
180                 else:
181                     nested_results.append(new_item)
182     return nested_results
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
```