

Optimizers

`docetl.optimizers.map_optimizer.optimizer.MapOptimizer`

A class for optimizing map operations in data processing pipelines.

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

Attributes:

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

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

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

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

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

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

```

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

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

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

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

Initialize the MapOptimizer.

Parameters:

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

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

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

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

Optimize the given operation configuration for the input data. Uses a staged evaluation approach: 1. For data exceeding limits: Try all plan types at once 2. For data within limits: - First try gleaning/proj synthesis - Compare with baseline - Selectively try chunking plans based on initial results

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

```

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

```

```

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

```

should_optimize(op_config, input_data)

Determine if the given operation configuration should be optimized.

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

```

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

```

`docetl.optimizers.reduce_optimizer.ReduceOptimizer`

A class that optimizes reduce operations in data processing pipelines.

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

Attributes:

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

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

1052         prompt = f"""
1053         For the {method} sampling method, we need to determine which
1054         keys from the input data should be used for generating embeddings.
1055
1056         Input data keys:
1057         {'', '.join(sample_input[0].keys())}
1058
1059         Sample Input Data:
1060         {json.dumps(sample_input[0], indent=2)[:1000]}...
1061
1062         Based on the reduce operation prompt and the sample input
1063         data, which keys should be used for generating embeddings? Use keys that
1064         will create meaningful embeddings (i.e., not id-related keys).
1065         Provide your answer as a list of key names that is a subset
1066         of the input data keys. You should pick only the 1-3 keys that are
1067         necessary for generating meaningful embeddings, that have relatively
1068         short values.
1069         """
1070
1071         parameters = {
1072             "type": "object",
1073             "properties": {
1074                 "embedding_keys": {"type": "array", "items":
1075 {"type": "string"}},
1076                 "explanation": {"type": "string"},
1077             },
1078             "required": ["embedding_keys", "explanation"],
1079         }
1080
1081         response = self.llm_client.generate_rewrite(
1082             [{"role": "user", "content": prompt}],
1083             system_prompt,
1084             parameters,
1085         )
1086         result = json.loads(response.choices[0].message.content)
1087         # TODO: validate that these exist
1088         embedding_keys = result["embedding_keys"]
1089         for key in result["embedding_keys"]:
1090             if key not in sample_input[0]:
1091                 embedding_keys.remove(key)
1092
1093         if not embedding_keys:
1094             # Select the reduce key
1095             self.console.log(
1096                 "No embedding keys found, selecting reduce key for
1097 embedding key"
1098             )
1099             embedding_keys = (
1100                 op_config["reduce_key"]
1101                 if isinstance(op_config["reduce_key"], list)
1102                 else [op_config["reduce_key"]]
1103             )
1104
1105             value_sampling_config["embedding_keys"] = embedding_keys
1106
1107         if method == "sem_sim":
1108             # Determine query text
1109             prompt = f"""
1110             For the semantic similarity (sem_sim) sampling method, we
1111             need to determine the query text to compare against when selecting
1112             samples.

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1601         plans = []
1602
1603         # Generate multiple fold prompts
1604         max_retries = 5
1605         retry_count = 0
1606         fold_prompts = []
1607
1608         while retry_count < max_retries and not fold_prompts:
1609             try:
1610                 fold_prompts = self._synthesize_fold_prompts(
1611                     op_config,
1612                     sample_input,
1613                     sample_output,
1614                     num_prompts=self.num_fold_prompts,
1615                 )
1616                 fold_prompts = list(set(fold_prompts))
1617                 if not fold_prompts:
1618                     raise ValueError("No fold prompts generated")
1619             except Exception as e:
1620                 retry_count += 1
1621                 if retry_count == max_retries:
1622                     raise RuntimeError(
1623                         f"Failed to generate fold prompts after
1624 {max_retries} attempts: {str(e)}"
1625                     )
1626                 self.console.log(
1627                     f"Retry {retry_count}/{max_retries}: Failed to
1628 generate fold prompts. Retrying..."
1629                 )
1630
1631         for batch_size in batch_sizes:
1632             for fold_idx, fold_prompt in enumerate(fold_prompts):
1633                 plan = op_config.copy()
1634                 plan["fold_prompt"] = fold_prompt
1635                 plan["fold_batch_size"] = batch_size
1636                 plan["associative"] = is_associative
1637                 plan["name"] = f"
1638 {op_config['name']}_bs_{batch_size}_fp_{fold_idx}"
1639                 plans.append(plan)
1640
1641         return plans
1642
1643     def _calculate_compression_ratio(
1644         self,
1645         op_config: dict[str, Any],
1646         sample_input: list[dict[str, Any]],
1647         sample_output: list[dict[str, Any]],
1648     ) -> float:
1649         """
1650         Calculate the compression ratio of the reduce operation.
1651
1652         This method compares the size of the input data to the size of
1653         the output data
1654         to determine how much the data is being compressed by the reduce
1655         operation.
1656
1657         Args:
1658             op_config (dict[str, Any]): Configuration for the reduce
1659             operation.
1660             sample_input (list[dict[str, Any]]): Sample input data.
1661             sample_output (list[dict[str, Any]]): Sample output data.

```

```

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

```

```

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

```



```

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

```

```

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

```

```

1906         - {{{{ reduce_key }}}}: The key used for grouping in the
1907         reduce operation
1908
1909         Provide the fold prompt as a string.
1910         """
1911         response = self.llm_client.generate_rewrite(
1912             [{"role": "user", "content": prompt}],
1913             system_prompt,
1914             parameters,
1915         )
1916         fold_prompt =
1917         json.loads(response.choices[0].message.content)["fold_prompt"]
1918
1919         # Run the operation with the fold prompt
1920         # Create a temporary plan with the fold prompt
1921         temp_plan = op_config.copy()
1922         temp_plan["fold_prompt"] = fold_prompt
1923         temp_plan["fold_batch_size"] = min(
1924             len(sample_input), 2
1925         ) # Use a small batch size for testing
1926
1927         # Run the operation with the fold prompt
1928         try:
1929             self._run_operation(
1930                 temp_plan, sample_input[:
1931                 temp_plan["fold_batch_size"]]
1932             )
1933
1934             return fold_prompt
1935         except Exception as e:
1936             self.console.log(
1937                 f"[red]Error in agent-generated fold prompt: {e}
1938                 [/red]"
1939             )
1940
1941         # Create a default fold prompt that instructs folding
1942         new data into existing output
1943         fold_prompt = f"""Analyze this batch of data using the
1944         following instructions:
1945
1946         {original_prompt}
1947
1948         However, instead of starting fresh, fold your analysis into the existing
1949         output that has already been generated. The existing output is provided
1950         in the 'output' variable below:
1951
1952         {{{{ output }}}}}
1953
1954         Remember, you must fold the new data into the existing output, do not
1955         start fresh."""
1956         return fold_prompt
1957
1958         with ThreadPoolExecutor(max_workers=self.max_threads) as
1959         executor:
1960             fold_prompts = list(
1961                 executor.map(lambda _: generate_single_prompt(),
1962                 range(num_prompts))
1963             )
1964
1965         return fold_prompts

```

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

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

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

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

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

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

    plan_scores = []
    plan_outputs = {}

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

```

        evaluation_sample = random.sample(input_data, sample_size)

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

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

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

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

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

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

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

            self.console.log(merged_plan["merge_prompt"])

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

```

```

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

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

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

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

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

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

```

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

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

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

Args:

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

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

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

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

Returns:

tuple[

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

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

BaseOperation],

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

The method performs the following steps:

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

2. Validates the output using the validator prompt.

3. Calculates a score based on the validation results.

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

"""

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

if return_instance:

output, operation_instance = output

validation_result = self._validate_reduce_output(
plan, validation_inputs, output, validator_prompt
)

Calculate a score based on validation results

valid_count = sum(
1

for result in validation_result["validation_results"]

if result["is_correct"]
)

score = valid_count /
len(validation_result["validation_results"])

if return_instance:

return plan, score, output, operation_instance

else:

return plan, score, output

def _synthesize_merge_prompt(
self, plan: dict[str, Any], sample_outputs: list[dict[str, Any]]

) -> str:
"""

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

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

Args:

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

Returns:

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

The method performs the following steps:

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

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

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

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

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

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

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

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

3. Minimally deviate from the reduce and fold prompts

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

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

Output Schema:

```
{json.dumps(output_schema, indent=2)}
```

Provide the merge prompt as a string.

```
"""
```

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

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

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

Initialize the ReduceOptimizer.

Parameters:

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

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

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

```

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

```

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

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

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

Parameters:

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

Returns:

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

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

```

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

```

```

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

```

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

`docetl.optimizers.join_optimizer.JoinOptimizer`

Source code in docetl/optimizers/join_optimizer.py

```

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

```



```

72         {map_prompt}
73
74         Respond with 'Yes' if the prompt is explicitly
75         categorical, detailing a finite set of possible outputs.
76         Respond with 'No' if the prompt allows for open-ended or
77         non-categorical responses.
78         Provide a brief explanation for your decision.""",
79     },
80 ]
81
82     response = self.llm_client.generate_rewrite(
83         messages,
84         "You are an expert in analyzing natural language prompts for
85         data processing tasks.",
86         {
87             "type": "object",
88             "properties": {
89                 "is_categorical": {
90                     "type": "string",
91                     "enum": ["Yes", "No"],
92                     "description": "Whether the prompt is explicitly
93         categorical",
94                 },
95                 "explanation": {
96                     "type": "string",
97                     "description": "Brief explanation for the
98         decision",
99                 },
100             },
101             "required": ["is_categorical", "explanation"],
102         },
103     )
104
105     analysis = json.loads(response.choices[0].message.content)
106
107     self.console.log("[bold]Map Prompt Analysis:[/bold]")
108     self.console.log(f"Is Categorical:
109 {analysis['is_categorical']}")
110     self.console.log(f"Explanation: {analysis['explanation']}")
111
112     return analysis["is_categorical"].lower() == "yes",
113     analysis["explanation"]
114
115     def _determine_duplicate_keys(
116         self,
117         input_data: list[dict[str, Any]],
118         reduce_key: list[str],
119         map_prompt: str | None = None,
120     ) -> tuple[bool, str]:
121         # Prepare a sample of the input data for analysis
122         sample_size = min(10, len(input_data))
123         data_sample = random.sample(
124             [{rk: item[rk] for rk in reduce_key} for item in
125         input_data], sample_size
126         )
127
128         context_prefix = ""
129         if map_prompt:
130             context_prefix = f"For context, these values came out of a
131         pipeline with the following prompt:\n\n{map_prompt}\n\n"
132

```

```

133         messages = [
134             {
135                 "role": "user",
136                 "content": f"{context_prefix}I want to do a reduce
137 operation on these values, and I need to determine if there are semantic
138 duplicates in the data, where the strings are different but they
139 technically belong in the same group. Note that exact string duplicates
140 should not be considered here.\n\nHere's a sample of the data (showing
141 the '{reduce_key}' field(s)): {data_sample}\n\nBased on this '{context
142 and ' if map_prompt else ''}sample, are there likely to be such semantic
143 duplicates (not exact string matches) in the dataset? Respond with 'yes'
144 only if you think there are semantic duplicates, or 'no' if you don't
145 see evidence of semantic duplicates or if you only see exact string
146 duplicates.",
147             },
148         ]
149         response = self.llm_client.generate_rewrite(
150             messages,
151             "You are an expert data analyst. Analyze the given data
152 sample and determine if there are likely to be semantic duplicate values
153 that belong in the same group, even if the strings are different.",
154             {
155                 "type": "object",
156                 "properties": {
157                     "likely_duplicates": {
158                         "type": "string",
159                         "enum": ["Yes", "No"],
160                         "description": "Whether duplicates are likely to
161 exist in the full dataset",
162                     },
163                     "explanation": {
164                         "type": "string",
165                         "description": "Brief explanation for the
166 decision",
167                     },
168                 },
169                 "required": ["likely_duplicates", "explanation"],
170             },
171         )
172
173         analysis = json.loads(response.choices[0].message.content)
174
175         self.console.log(f"[bold]Duplicate Analysis for '{reduce_key}':
176 [/bold]")
177         self.console.log(f"Likely Duplicates:
178 {analysis['likely_duplicates']}")
179         self.console.log(f"Explanation: {analysis['explanation']}")
180
181         if analysis["likely_duplicates"].lower() == "yes":
182             self.console.log(
183                 "[yellow]Duplicates are likely. Consider using a
184 deduplication strategy in the resolution step.[/yellow]"
185             )
186             return True, analysis["explanation"]
187         return False, ""
188
189     def _sample_random_pairs(
190         self, input_data: list[dict[str, Any]], n: int
191     ) -> list[tuple[int, int]]:
192         """Sample random pairs of indices, excluding exact matches."""
193         pairs = set()

```

```

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

```

```

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

```

```

316
317     Please generate the comparison prompt, which should be a Jinja2
318     template:
319     """
320         }
321     ]
322
323     response = self.llm_client.generate_rewrite(
324         messages,
325         system_prompt,
326         {
327             "type": "object",
328             "properties": {
329                 "comparison_prompt": {
330                     "type": "string",
331                     "description": "Detailed comparison prompt for
332 entity resolution",
333                 }
334             },
335             "required": ["comparison_prompt"],
336         },
337     )
338
339     comparison_prompt =
340     json.loads(response.choices[0].message.content)[
341         "comparison_prompt"
342     ]
343
344     # Log the synthesized comparison prompt
345     self.console.log("[green]Synthesized comparison prompt:
346 [/green]")
347     self.console.log(comparison_prompt)
348
349     if not comparison_prompt:
350         raise ValueError(
351             "Could not synthesize a comparison prompt. Please
352 provide a comparison prompt in the config."
353         )
354
355     return comparison_prompt
356
357     def synthesize_resolution_prompt(
358         self,
359         map_prompt: str | None,
360         reduce_key: list[str],
361         output_schema: dict[str, str],
362     ) -> str:
363         system_prompt = f"""You are an AI assistant tasked with creating
364 a resolution prompt for LLM-assisted entity resolution.
365         Your task is to create a prompt that will be used to merge
366 multiple duplicate keys into a single, consolidated key.
367         The key(s) being resolved (known as the reduce_key) are {'',
368 '.join(reduce_key)}.
369         The duplicate keys will be provided in a list called 'inputs' in
370 a Jinja2 template.
371         """
372
373         if map_prompt:
374             system_prompt += f"\n\nFor context, here is the prompt used
375 earlier in the pipeline to create the inputs to resolve: {map_prompt}"
376

```

```

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

```

```

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

```

```

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

```



```

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

```

```

621         )
622
623         if blocking_rules:
624             false_negatives, rule_selectivity =
625 self._verify_blocking_rule(
626             input_data,
627             blocking_rules[0],
628             blocking_keys,
629             comparison_results,
630         )
631         # If more than 50% of the sample is false negatives, reject
632 the blocking rule
633         if len(false_negatives) > len(sampled_pairs) / 2:
634             if false_negatives:
635                 self.console.log(
636                     f"[red]Blocking rule rejected.
637 {len(false_negatives)} false negatives detected in the sample
638 ({len(false_negatives) / len(sampled_pairs):.2f} of the sample).[/red]"
639                 )
640                 for i, j in false_negatives[:5]: # Show up to 5
641 examples
642                     self.console.log(
643                         f"  Filtered pair: {{ {blocking_keys[0]}}:
644 {input_data[i][blocking_keys[0]]} }} and {{ {blocking_keys[0]}}:
645 {input_data[j][blocking_keys[0]]} }}"
646                     )
647                     if len(false_negatives) > 5:
648                         self.console.log(f"  ... and
649 {len(false_negatives) - 5} more.")
650                     blocking_rules = (
651                         []
652                     ) # Clear the blocking rule if it introduces false
653 negatives or is too selective
654                     elif not false_negatives and rule_selectivity >
655 estimated_selectivity:
656                         self.console.log(
657                             "[green]Blocking rule verified. No false negatives
658 detected in the sample and selectivity is within estimated selectivity.
659 [/green]"
660                         )
661                     else:
662                         # TODO: ask user if they want to use the blocking rule,
663 or come up with some good default behavior
664                         blocking_rules = []
665
666             optimized_config = self._update_config(threshold, blocking_keys,
667 blocking_rules)
668             return optimized_config, embedding_cost + comparison_cost
669
670         def optimize_equipjoin(
671             self,
672             left_data: list[dict[str, Any]],
673             right_data: list[dict[str, Any]],
674             skip_map_gen: bool = False,
675             skip_containment_gen: bool = False,
676         ) -> tuple[dict[str, Any], float, dict[str, Any]]:
677             left_keys = self.op_config.get("blocking_keys", {}).get("left",
678 [])
679             right_keys = self.op_config.get("blocking_keys",
680 {}).get("right", [])
681

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

    )
    self.console.log(
        f"[bold cyan]|[/bold cyan] [yellow]Target Recall:[/yellow]
{self.target_recall:.0%}"
    )
    self.console.log(
        f"[bold cyan]|[/bold cyan] [yellow]Estimate:[/yellow]
{selectivity_estimate:.4f}"
    )
    self.console.log(
        "[bold
cyan]"
    )

```



```

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

```

```

1597     evaluation."""",
1598         }
1599     ]
1600
1601     for attempt in range(self.agent_max_retries): # Up to 3
1602         attempts
1603         # Generate blocking rule using the LLM
1604         response = self.llm_client.generate_rewrite(
1605             messages,
1606             "You are an expert in entity resolution and Python
1607 programming. Your task is to generate one efficient blocking rule based
1608 on the given sample comparisons and data structure.",
1609             {
1610                 "type": "object",
1611                 "properties": {
1612                     "blocking_rule": {
1613                         "type": "string",
1614                         "description": "One-line Python statement
1615 acting as a blocking rule",
1616                     }
1617                 },
1618                 "required": ["blocking_rule"],
1619             },
1620         )
1621
1622         # Extract the blocking rule from the LLM response
1623         blocking_rule = response.choices[0].message.content
1624         blocking_rule =
1625 json.loads(blocking_rule).get("blocking_rule")
1626
1627         if blocking_rule:
1628             self.console.log("") # Print a newline
1629
1630             if blocking_rule.strip() == "True":
1631                 self.console.log(
1632                     "[yellow]No suitable blocking rule could be
1633 found. Proceeding without a blocking rule.[/yellow]"
1634                 )
1635                 return []
1636
1637                 self.console.log(
1638                     f"[bold]Generated blocking rule (Attempt {attempt +
1639 1}):[/bold] {blocking_rule}"
1640                 )
1641
1642                 # Test the blocking rule
1643                 filtered_pairs = self._test_blocking_rule(
1644                     input_data, blocking_keys, blocking_rule,
1645 comparisons
1646                 )
1647
1648                 if not filtered_pairs:
1649                     self.console.log(
1650                         "[green]Blocking rule looks good! No known
1651 matches were filtered out.[/green]"
1652                     )
1653                     return [blocking_rule]
1654                 else:
1655                     feedback = f"The previous rule incorrectly filtered
1656 out {len(filtered_pairs)} known matches. "
1657                     feedback += (

```

```

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

```

```

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

```

```

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

```

```

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

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

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

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

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

```

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

Example structure of a one-line blocking rule:

```
"(condition1) or (condition2) or (condition3)"
```

Where conditions could be comparisons like:

```
"left['{left_keys[0]}'].lower() == right['{right_keys[0]}'].lower()"
"abs(len(left['{left_keys[0]}']) - len(right['{right_keys[0]}'])) <=
5"
"any(word in left['{left_keys[0]}'].lower() for word in
right['{right_keys[0]}'].lower().split())"
```

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

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

```
    }
]

for attempt in range(self.agent_max_retries):
    response = self.llm_client.generate_rewrite(
        messages,
        "You are an expert in entity resolution and Python
programming. Your task is to generate one efficient blocking rule based
on the given sample comparisons and data structure.",
        {
            "type": "object",
            "properties": {
                "blocking_rule": {
                    "type": "string",
                    "description": "One-line Python statement
acting as a blocking rule",
                }
            },
            "required": ["blocking_rule"],
        },
    )

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

    if blocking_rule:
```

```

        self.console.log("")

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

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

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

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

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

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

```



```

        return []

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

        filtered_pairs = []

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

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

        return filtered_pairs

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

```

```

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

        false_negatives = []
        total_pairs = 0
        blocked_pairs = 0

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

        rule_selectivity = blocked_pairs / total_pairs if total_pairs >
0 else 0

        return false_negatives, rule_selectivity

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

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

```

```

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

        false_negatives = []
        total_pairs = 0
        blocked_pairs = 0

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

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

        rule_selectivity = blocked_pairs / total_pairs if total_pairs >
0 else 0

        return false_negatives, rule_selectivity

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

```

should_optimize(input_data)

Determine if the given operation configuration should be optimized.

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

```

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

```

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