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

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

```python
class 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:
        config (dict[str, Any]): The configuration dictionary for the
optimizer.
        console (Console): A Rich console object for pretty printing.
        llm_client (LLMClient): A client for interacting with a language
model.
        _run_operation (Callable): A function to execute operations.
        max_threads (int): The maximum number of threads to use for
parallel execution.
        timeout (int): The timeout in seconds for operation execution.

    """

    def __init__(
        self,
        runner,
        run_operation: Callable,
        timeout: int = 10,
        is_filter: bool = False,
        depth: int = 1,
    ):
        """
        Initialize the MapOptimizer.

        Args:
            runner (Runner): The runner object.
            run_operation (Callable): A function to execute operations.
            timeout (int, optional): The timeout in seconds for operation
execution. Defaults to 10.
            is_filter (bool, optional): If True, the operation is a
filter operation. Defaults to False.
        """
        self.runner = runner
        self.config = runner.config
        self.console = runner.console
        self.llm_client = runner.optimizer.llm_client
        self._run_operation = run_operation
        self.max_threads = runner.max_threads
        self.timeout = runner.optimizer.timeout
        self._num_plans_to_evaluate_in_parallel = 5
        self.is_filter = is_filter
        self.k_to_pairwise_compare = 6

        self.plan_generator = PlanGenerator(
            runner,
            self.llm_client,
            self.console,
            self.config,
            run_operation,
            self.max_threads,
            is_filter,
            depth,
        )
        self.evaluator = Evaluator(
            self.llm_client,
            self.console,
            self._run_operation,
            self.timeout,
            self._num_plans_to_evaluate_in_parallel,
            self.is_filter,
        )
        self.prompt_generator = PromptGenerator(
            self.runner,
            self.llm_client,
            self.console,
            self.config,
            self.max_threads,
            self.is_filter,
        )

    def should_optimize(
        self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
    ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]]]:
        """
        Determine if the given operation configuration should be
optimized.
        """
        (
            input_data,
            output_data,
            _,
            _,
            validator_prompt,
            assessment,
            data_exceeds_limit,
        ) = self._should_optimize_helper(op_config, input_data)
        if data_exceeds_limit or assessment.get("needs_improvement",
True):
            assessment_str = (
                "\n".join(assessment.get("reasons", []))
                + "\n\nHere are some improvements that may help:\n"
                + "\n".join(assessment.get("improvements", []))
            )
            if data_exceeds_limit:
                assessment_str += "\nAlso, the input data exceeds the
token limit."
            return assessment_str, input_data, output_data
        else:
            return "", input_data, output_data

    def _should_optimize_helper(
        self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
    ) -> tuple[
        list[dict[str, Any]],
        list[dict[str, Any]],
        int,
        float,
        str,
        dict[str, Any],
        bool,
    ]:
        """
        Determine if the given operation configuration should be
optimized.
        Create a custom validator prompt and assess the operation's
performance
        using the validator.
        """
        self.console.post_optimizer_status(StageType.SAMPLE_RUN)
        input_data = copy.deepcopy(input_data)
        # Add id to each input_data
        for i in range(len(input_data)):
            input_data[i]["_map_opt_id"] = str(uuid.uuid4())

        # Define the token limit (adjust as needed)
        model_input_context_length = model_cost.get(
            op_config.get("model", self.config.get("default_model")), {}
        ).get("max_input_tokens", 8192)

        # Render the prompt with all sample inputs and count tokens
        total_tokens = 0
        exceed_count = 0
        for sample in input_data:
            rendered_prompt =
Template(op_config["prompt"]).render(input=sample)
            prompt_tokens = count_tokens(
                rendered_prompt,
                op_config.get("model", self.config.get("default_model")),
            )
            total_tokens += prompt_tokens

            if prompt_tokens > model_input_context_length:
                exceed_count += 1

        # Calculate average tokens and percentage of samples exceeding
limit
        avg_tokens = total_tokens / len(input_data)
        exceed_percentage = (exceed_count / len(input_data)) * 100

        data_exceeds_limit = exceed_count > 0
        if exceed_count > 0:
            self.console.log(
                f"[yellow]Warning: {exceed_percentage:.2f}% of prompts
exceed token limit. "
                f"Average token count: {avg_tokens:.2f}. "
                f"Truncating input data when generating validators.
[/yellow]"
            )

        # Execute the original operation on the sample data
        no_change_start = time.time()
        output_data = self._run_operation(op_config, input_data,
is_build=True)
        no_change_runtime = time.time() - no_change_start

        # Capture output for the sample run
        self.runner.optimizer.captured_output.save_optimizer_output(
            stage_type=StageType.SAMPLE_RUN,
            output={
                "operation_config": op_config,
                "input_data": input_data,
                "output_data": output_data,
            },
        )

        # Generate custom validator prompt
        self.console.post_optimizer_status(StageType.SHOULD_OPTIMIZE)
        validator_prompt =
self.prompt_generator._generate_validator_prompt(
            op_config, input_data, output_data
        )

        # Log the validator prompt
        self.console.log("[bold]Validator Prompt:[/bold]")
        self.console.log(validator_prompt)
        self.console.log("\n")  # Add a newline for better readability

        # Step 2: Use the validator prompt to assess the operation's
performance
        assessment = self.evaluator._assess_operation(
            op_config, input_data, output_data, validator_prompt
        )

        # Print out the assessment
        self.console.log(
            f"[bold]Assessment for whether we should improve operation
{op_config['name']}:[/bold]"
        )
        for key, value in assessment.items():
            self.console.log(f"[bold cyan]{key}:[/bold cyan] [yellow]
{value}[/yellow]")
        self.console.log("\n")  # Add a newline for better readability

        self.runner.optimizer.captured_output.save_optimizer_output(
            stage_type=StageType.SHOULD_OPTIMIZE,
            output={
                "validator_prompt": validator_prompt,
                "needs_improvement": assessment.get("needs_improvement",
True),
                "reasons": assessment.get("reasons", []),
                "improvements": assessment.get("improvements", []),
            },
        )
        self.console.post_optimizer_rationale(
            assessment.get("needs_improvement", True),
            "\n".join(assessment.get("reasons", []))
            + "\n\n"
            + "\n".join(assessment.get("improvements", [])),
            validator_prompt,
        )

        return (
            input_data,
            output_data,
            model_input_context_length,
            no_change_runtime,
            validator_prompt,
            assessment,
            data_exceeds_limit,
        )

```

```python
    def optimize(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        plan_types: list[str] | None = ["chunk", "proj_synthesis", "glean"],
    ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
        """
        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
        """
        # Verify that the plan types are valid
        for plan_type in plan_types:
            if plan_type not in ["chunk", "proj_synthesis", "glean"]:
                raise ValueError(
                    f"Invalid plan type: {plan_type}. Valid plan types are: chunk, proj_synthesis, glean."
                )

        (
            input_data,
            output_data,
            model_input_context_length,
            no_change_runtime,
            validator_prompt,
            assessment,
            data_exceeds_limit,
        ) = self._should_optimize_helper(op_config, input_data)

        if not self.config.get("optimizer_config", {}).get("force_decompose", False):
            if not data_exceeds_limit and not assessment.get("needs_improvement", True):
                self.console.log(
                    f"[green]No improvement needed for operation {op_config['name']}[/green]"
                )
                return (
                    [op_config],
                    output_data,
                    self.plan_generator.subplan_optimizer_cost,
                )

        # Select consistent evaluation samples
        num_evaluations = min(5, len(input_data))
        evaluation_samples = select_evaluation_samples(input_data, num_evaluations)

        if data_exceeds_limit:
            # For data exceeding limits, try all plan types at once
            return self._evaluate_all_plans(
                op_config,
                input_data,
                evaluation_samples,
                validator_prompt,
                plan_types,
                model_input_context_length,
                data_exceeds_limit=True,
            )

        # For data within limits, use staged evaluation
        return self._staged_evaluation(
            op_config,
            input_data,
            evaluation_samples,
            validator_prompt,
            plan_types,
            no_change_runtime,
            model_input_context_length,
        )

    def _select_best_plan(
        self,
        results: dict[str, tuple[float, float, list[dict[str, Any]]]],
        op_config: dict[str, Any],
        evaluation_samples: list[dict[str, Any]],
        validator_prompt: str,
        candidate_plans: dict[str, list[dict[str, Any]]],
    ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], str, dict[str,
int]]:
        """
        Select the best plan from evaluation results using top-k
comparison.

        Returns:
            Tuple of (best plan, best output, best plan name, pairwise
rankings)
        """
        # Sort results by score in descending order
        sorted_results = sorted(results.items(), key=lambda x: x[1][0],
reverse=True)

        # Take the top k plans
        top_plans = sorted_results[: self.k_to_pairwise_compare]

        # Check if there are no top plans
        if len(top_plans) == 0:
            raise ValueError(
                "No valid plans were generated. Unable to proceed with
optimization."
            )

        # Include any additional plans that are tied with the last plan
        tail_score = (
            top_plans[-1][1][0]
            if len(top_plans) == self.k_to_pairwise_compare
            else float("-inf")
        )
        filtered_results = dict(
            top_plans
            + [
                item
                for item in sorted_results[len(top_plans) :]
                if item[1][0] == tail_score
            ]
        )

        # Perform pairwise comparisons on filtered plans
        if len(filtered_results) > 1:
            pairwise_rankings = self.evaluator._pairwise_compare_plans(
                filtered_results, validator_prompt, op_config,
evaluation_samples
            )
            best_plan_name = max(pairwise_rankings,
key=pairwise_rankings.get)
        else:
            pairwise_rankings = {k: 0 for k in results.keys()}
            best_plan_name = next(iter(filtered_results))

        # Display results table
        self.console.log(
            f"\n[bold]Plan Evaluation Results for {op_config['name']}
({op_config['type']}, {len(results)} plans, {len(evaluation_samples)}
samples):[/bold]"
        )
        table = Table(show_header=True, header_style="bold magenta")
        table.add_column("Plan", style="dim")
        table.add_column("Score", justify="right", width=10)
        table.add_column("Runtime", justify="right", width=10)
        table.add_column("Pairwise Wins", justify="right", width=10)

        for plan_name, (score, runtime, _) in sorted_results:
            table.add_row(
                plan_name,
                f"{score:.2f}",
                f"{runtime:.2f}s",
                f"{pairwise_rankings.get(plan_name, 0)}",
            )

        self.console.log(table)
        self.console.log("\n")

        try:
            best_plan = candidate_plans[best_plan_name]
            best_output = results[best_plan_name][2]
        except KeyError:
            raise ValueError(
                f"Best plan name {best_plan_name} not found in candidate
plans. Candidate plan names: {candidate_plans.keys()}"
            )

        self.console.log(
            f"[green]Current best plan: {best_plan_name} for operation
{op_config['name']} "
            f"(Score: {results[best_plan_name][0]:.2f}, "
            f"Runtime: {results[best_plan_name][1]:.2f}s)[/green]"
        )

        return best_plan, best_output, best_plan_name, pairwise_rankings

    def _staged_evaluation(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        evaluation_samples: list[dict[str, Any]],
        validator_prompt: str,
        plan_types: list[str],
        no_change_runtime: float,
        model_input_context_length: int,
    ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
        """Stage 1: Try gleaning and proj synthesis plans first"""
        candidate_plans = {"no_change": [op_config]}

        # Generate initial plans (gleaning and proj synthesis)
        if "glean" in plan_types:
            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)

        if "proj_synthesis" in plan_types and 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)

        # Evaluate initial plans
        initial_results = self._evaluate_plans(
            candidate_plans,
            op_config,
            evaluation_samples,
            validator_prompt,
            no_change_runtime,
        )

        # Get best initial plan
        best_plan, best_output, best_plan_name, pairwise_rankings = (
            self._select_best_plan(
                initial_results,
                op_config,
                evaluation_samples,
                validator_prompt,
                candidate_plans,
            )
        )
        best_is_better_than_baseline = best_plan_name != "no_change"

        # Stage 2: Decide whether/how to try chunking plans
        if "chunk" in plan_types:
            if best_is_better_than_baseline:
                # Try 2 random chunking plans first
                self.console.log(
                    "[bold magenta]Trying sample of chunking plans...[/bold magenta]"
                )
                chunk_plans = self.plan_generator._generate_chunk_size_plans(
                    op_config, input_data, validator_prompt, model_input_context_length
                )

                if chunk_plans:
                    # Sample 2 random plans
                    chunk_items = list(chunk_plans.items())
                    sample_plans = dict(
                        random.sample(chunk_items, min(2, len(chunk_items)))
                    )
                    sample_results = self._evaluate_plans(
                        sample_plans, op_config, evaluation_samples, validator_prompt
                    )

                    # Do pairwise comparison between sampled plans and current best
                    current_best = {best_plan_name: initial_results[best_plan_name]}
                    current_best.update(sample_results)

                    _, _, new_best_name, new_pairwise_rankings = self._select_best_plan(
                        current_best,
                        op_config,
                        evaluation_samples,
                        validator_prompt,
                        {**{best_plan_name: best_plan}, **sample_plans},
                    )

                    if new_best_name == best_plan_name:
                        self.console.log(
                            "[yellow]Sample chunking plans did not improve results. Keeping current best plan.[/yellow]"
                        )
                        return (
                            best_plan,
                            best_output,
                            self.plan_generator.subplan_optimizer_cost,
                        )

                    # If a sampled plan wins, evaluate all chunking plans
                    self.console.log(
                        "[bold magenta]Generating all chunking plans...[/bold magenta]"
                    )
                    chunk_results = self._evaluate_plans(
                        chunk_plans, op_config, evaluation_samples, validator_prompt
                    )
                    initial_results.update(chunk_results)
                    candidate_plans.update(chunk_plans)
            else:
                # Try all chunking plans since no improvement found yet
                self.console.log(
                    "[bold magenta]Generating chunking plans...[/bold
magenta]"
                )
                chunk_plans =
self.plan_generator._generate_chunk_size_plans(
                    op_config, input_data, validator_prompt,
model_input_context_length
                )
                chunk_results = self._evaluate_plans(
                    chunk_plans, op_config, evaluation_samples,
validator_prompt
                )
                initial_results.update(chunk_results)
                candidate_plans.update(chunk_plans)

        # Final selection of best plan
        best_plan, best_output, _, final_pairwise_rankings =
self._select_best_plan(
            initial_results,
            op_config,
            evaluation_samples,
            validator_prompt,
            candidate_plans,
        )

        # Capture evaluation results with pairwise rankings
        ratings = {k: v[0] for k, v in initial_results.items()}
        runtime = {k: v[1] for k, v in initial_results.items()}
        sample_outputs = {k: v[2] for k, v in initial_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": final_pairwise_rankings,
            },
        )

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

    def _evaluate_plans(
        self,
        plans: dict[str, list[dict[str, Any]]],
        op_config: dict[str, Any],
        evaluation_samples: list[dict[str, Any]],
        validator_prompt: str,
        no_change_runtime: float | None = None,
    ) -> dict[str, tuple[float, float, list[dict[str, Any]]]]:
        """Helper method to evaluate a set of plans in parallel"""
        results = {}
        plans_list = list(plans.items())

        for i in range(0, len(plans_list),
self._num_plans_to_evaluate_in_parallel):
            batch = plans_list[i : i +
self._num_plans_to_evaluate_in_parallel]
            with ThreadPoolExecutor(
                max_workers=self._num_plans_to_evaluate_in_parallel
            ) as executor:
                futures = {
                    executor.submit(
                        self.evaluator._evaluate_plan,
                        plan_name,
                        op_config,
                        plan,
                        copy.deepcopy(evaluation_samples),
                        validator_prompt,
                    ): plan_name
                    for plan_name, plan in batch
                }
                for future in as_completed(futures):
                    plan_name = futures[future]
                    try:
                        score, runtime, output =
future.result(timeout=self.timeout)
                        results[plan_name] = (score, runtime, output)
                    except concurrent.futures.TimeoutError:
                        self.console.log(
                            f"[yellow]Plan {plan_name} timed out and will
be skipped.[/yellow]"
                        )
                    except Exception as e:
                        self.console.log(
                            f"[red]Error in plan {plan_name}: {str(e)}
[/red]"
                        )

        if "no_change" in results and no_change_runtime is not None:
            results["no_change"] = (
                results["no_change"][0],
                no_change_runtime,
                results["no_change"][2],
            )

        return results

    def _evaluate_all_plans(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        evaluation_samples: list[dict[str, Any]],
        validator_prompt: str,
        plan_types: list[str],
        model_input_context_length: int,
        data_exceeds_limit: bool,
    ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
        """
        Evaluate all plans for a given operation configuration.
        """
        candidate_plans = {}

        # Generate all plans
        self.console.post_optimizer_status(StageType.CANDIDATE_PLANS)
        self.console.log(
            f"[bold magenta]Generating {len(plan_types)} plans...[/bold
magenta]"
```

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

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

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

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

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

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

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

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

Initialize the MapOptimizer.

Parameters:

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

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

```python
def __init__(
    self,
    runner,
    run_operation: Callable,
    timeout: int = 10,
    is_filter: bool = False,
    depth: int = 1,
):
    """
    Initialize the MapOptimizer.

    Args:
        runner (Runner): The runner object.
        run_operation (Callable): A function to execute operations.
        timeout (int, optional): The timeout in seconds for operation execution. Defaults to 10.
        is_filter (bool, optional): If True, the operation is a filter operation. Defaults to False.
    """
    self.runner = runner
    self.config = runner.config
    self.console = runner.console
    self.llm_client = runner.optimizer.llm_client
    self._run_operation = run_operation
    self.max_threads = runner.max_threads
    self.timeout = runner.optimizer.timeout
    self._num_plans_to_evaluate_in_parallel = 5
    self.is_filter = is_filter
    self.k_to_pairwise_compare = 6

    self.plan_generator = PlanGenerator(
        runner,
        self.llm_client,
        self.console,
        self.config,
        run_operation,
        self.max_threads,
        is_filter,
        depth,
    )
    self.evaluator = Evaluator(
        self.llm_client,
        self.console,
        self._run_operation,
        self.timeout,
        self._num_plans_to_evaluate_in_parallel,
        self.is_filter,
    )
    self.prompt_generator = PromptGenerator(
        self.runner,
        self.llm_client,
        self.console,
        self.config,
        self.max_threads,
        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`

```python
def optimize(
    self,
    op_config: dict[str, Any],
    input_data: list[dict[str, Any]],
    plan_types: list[str] | None = ["chunk", "proj_synthesis", "glean"],
) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
    """
    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
    """
    # Verify that the plan types are valid
    for plan_type in plan_types:
        if plan_type not in ["chunk", "proj_synthesis", "glean"]:
            raise ValueError(
                f"Invalid plan type: {plan_type}. Valid plan types are:
chunk, proj_synthesis, glean."
            )

    (
        input_data,
        output_data,
        model_input_context_length,
        no_change_runtime,
        validator_prompt,
        assessment,
        data_exceeds_limit,
    ) = self._should_optimize_helper(op_config, input_data)

    if not self.config.get("optimizer_config", {}).get("force_decompose",
False):
        if not data_exceeds_limit and not
assessment.get("needs_improvement", True):
            self.console.log(
                f"[green]No improvement needed for operation
{op_config['name']}[/green]"
            )
            return (
                [op_config],
                output_data,
                self.plan_generator.subplan_optimizer_cost,
            )

    # Select consistent evaluation samples
    num_evaluations = min(5, len(input_data))
    evaluation_samples = select_evaluation_samples(input_data,
num_evaluations)

    if data_exceeds_limit:
        # For data exceeding limits, try all plan types at once
        return self._evaluate_all_plans(
            op_config,
            input_data,
            evaluation_samples,
            validator_prompt,
            plan_types,
            model_input_context_length,
            data_exceeds_limit=True,
        )

    # For data within limits, use staged evaluation
    return self._staged_evaluation(
        op_config,
        input_data,
        evaluation_samples,
        validator_prompt,
        plan_types,
        no_change_runtime,
        model_input_context_length,
    )
```

`should_optimize(op_config, input_data)`

Determine if the given operation configuration should be optimized.

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

```
def should_optimize(
    self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]]]:
    """
    Determine if the given operation configuration should be optimized.
    """
    (
        input_data,
        output_data,
        _,
        _,
        validator_prompt,
        assessment,
        data_exceeds_limit,
    ) = self._should_optimize_helper(op_config, input_data)
    if data_exceeds_limit or assessment.get("needs_improvement", True):
        assessment_str = (
            "\n".join(assessment.get("reasons", []))
            + "\n\nHere are some improvements that may help:\n"
            + "\n".join(assessment.get("improvements", []))
        )
        if data_exceeds_limit:
            assessment_str += "\nAlso, the input data exceeds the token
limit."
        return assessment_str, input_data, output_data
    else:
        return "", input_data, output_data
```

`docetl.optimizers.reduce_optimizer.ReduceOptimizer`

A class that optimizes reduce operations in data processing pipelines.

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

Attributes:

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

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

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

    def __init__(
        self,
        runner,
        run_operation: Callable,
        num_fold_prompts: int = 1,
        num_samples_in_validation: int = 10,
    ):
        """
        Initialize the ReduceOptimizer.

        Args:
            config (dict[str, Any]): Configuration dictionary for the
optimizer.
            console (Console): Rich console object for pretty printing.
            llm_client (LLMClient): Client for interacting with a
language model.
            max_threads (int): Maximum number of threads to use for
parallel processing.
            run_operation (Callable): Function to run an operation.
            num_fold_prompts (int, optional): Number of fold prompts to
generate. Defaults to 1.
            num_samples_in_validation (int, optional): Number of samples
to use in validation. Defaults to 10.
        """
        self.runner = runner
        self.config = self.runner.config
        self.console = self.runner.console
        self.llm_client = self.runner.optimizer.llm_client
        self._run_operation = run_operation
        self.max_threads = self.runner.max_threads
        self.num_fold_prompts = num_fold_prompts
        self.num_samples_in_validation = num_samples_in_validation
        self.status = self.runner.status

    def should_optimize_helper(
        self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
    ) -> str:
        # Check if we're running out of token limits for the reduce prompt
        model = op_config.get("model", self.config.get("default_model", "gpt-4o-mini"))
        model_input_context_length = model_cost.get(model, {}).get(
            "max_input_tokens", 4096
        )

        # Find the key with the longest value
        if op_config["reduce_key"] == ["_all"]:
            sample_key = tuple(["_all"])
        else:
            longest_key = max(
                op_config["reduce_key"], key=lambda k: len(str(input_data[0][k]))
            )
            sample_key = tuple(
                input_data[0][k] if k == longest_key else input_data[0][k]
                for k in op_config["reduce_key"]
            )

        # Render the prompt with a sample input
        prompt_template = Template(op_config["prompt"])
        sample_prompt = prompt_template.render(
            reduce_key=dict(zip(op_config["reduce_key"], sample_key)),
            inputs=[input_data[0]],
        )

        # Count tokens in the sample prompt
        prompt_tokens = count_tokens(sample_prompt, model)

        self.console.post_optimizer_status(StageType.SAMPLE_RUN)
        original_output = self._run_operation(op_config, input_data)

        # Step 1: Synthesize a validator prompt
        self.console.post_optimizer_status(StageType.SHOULD_OPTIMIZE)
        validator_prompt = self._generate_validator_prompt(
            op_config, input_data, original_output
        )

        # Log the validator prompt
        self.console.log("[bold]Validator Prompt:[/bold]")
        self.console.log(validator_prompt)
        self.console.log("\n")  # Add a newline for better readability

        # Step 2: validate the output
        validator_inputs = self._create_validation_inputs(
            input_data, op_config["reduce_key"]
        )
        validation_results = self._validate_reduce_output(
            op_config, validator_inputs, original_output, validator_prompt
        )

        return (
            validation_results,
            prompt_tokens,
            model_input_context_length,
            model,
            validator_prompt,
            original_output,
        )

    def should_optimize(
        self, op_config: dict[str, Any], input_data: list[dict[str,
Any]]
    ) -> tuple[str, list[dict[str, Any]], list[dict[str, Any]]]:
        (
            validation_results,
            prompt_tokens,
            model_input_context_length,
            model,
            validator_prompt,
            original_output,
        ) = self.should_optimize_helper(op_config, input_data)
        if prompt_tokens * 1.5 > model_input_context_length:
            return (
                "The reduce prompt is likely to exceed the token limit
for model {model}.",
                input_data,
                original_output,
            )

        if validation_results.get("needs_improvement", False):
            return (
                "\n".join(
                    [
                        f"Issues: {result['issues']} Suggestions:
{result['suggestions']}"
                        for result in
validation_results["validation_results"]
                    ]
                ),
                input_data,
                original_output,
            )
        else:
            return "", input_data, original_output

    def optimize(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        level: int = 1,
    ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
        """
        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)

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.

        Returns:
            tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A
tuple containing the list of optimized configurations
            and the list of outputs from the optimized operation(s), and
the cost of the operation due to synthesizing any resolve operations.
        """
        (
            validation_results,
            prompt_tokens,
            model_input_context_length,
            model,
            validator_prompt,
            original_output,
        ) = self.should_optimize_helper(op_config, input_data)

        # add_map_op = False
        if prompt_tokens * 2 > model_input_context_length:
            # add_map_op = True
            self.console.log(
                f"[yellow]Warning: The reduce prompt exceeds the token
limit for model {model}. "
                f"Token count: {prompt_tokens}, Limit:
{model_input_context_length}. "
                f"Add a map operation to the pipeline.[/yellow]"
            )

        # # Also query an agent to look at a sample of the inputs and
see if they think a map operation would be helpful
        # preprocessing_steps = ""
        # should_use_map, preprocessing_steps = self._should_use_map(
        #     op_config, input_data
        # )
        # if should_use_map or add_map_op:
        #     # Synthesize a map operation
        #     map_prompt, map_output_schema =
self._synthesize_map_operation(
        #         op_config, preprocessing_steps, input_data
        #     )
        #     # Change the reduce operation prompt to use the map schema
        #     new_reduce_prompt =
self._change_reduce_prompt_to_use_map_schema(
        #         op_config["prompt"], map_output_schema
        #     )
        #     op_config["prompt"] = new_reduce_prompt

        #     # Return unoptimized map and reduce operations
        #     return [map_prompt, op_config], input_data, 0.0

        # Print the validation results
        self.console.log("[bold]Validation Results on Initial Sample:
[/bold]")
```

```
        if validation_results["needs_improvement"] or self.config.get(
            "optimizer_config", {}
        ).get("force_decompose", False):
            self.console.post_optimizer_rationale(
                should_optimize=True,
                rationale="\n".join(
                    [
                        f"Issues: {result['issues']} Suggestions: {result['suggestions']}"
                        for result in validation_results["validation_results"]
                    ]
                ),
                validator_prompt=validator_prompt,
            )
            self.console.log(
                "\n".join(
                    [
                        f"Issues: {result['issues']} Suggestions: {result['suggestions']}"
                        for result in validation_results["validation_results"]
                    ]
                )
            )

            # Step 3: Evaluate if decomposition is beneficial
            decomposition_result = self._evaluate_decomposition(
                op_config, input_data, level
            )

            if decomposition_result["should_decompose"]:
                return self._optimize_decomposed_reduce(
                    decomposition_result, op_config, input_data, level
                )

            return self._optimize_single_reduce(op_config, input_data, validator_prompt)
        else:
            self.console.log(f"No improvements identified; {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

    def _should_use_map(
        self, op_config: dict[str, Any], input_data: list[dict[str, Any]]
    ) -> tuple[bool, str]:
        """
        Determine if a map operation should be used based on the input data.
        """
        # Sample a random input item
        sample_input = random.choice(input_data)

        # Format the prompt with the sample input
        prompt_template = Template(op_config["prompt"])
        formatted_prompt = prompt_template.render(
            reduce_key=dict(
                zip(op_config["reduce_key"],
sample_input[op_config["reduce_key"]])
            ),
            inputs=[sample_input],
        )

        # Prepare the message for the LLM
        messages = [{"role": "user", "content": formatted_prompt}]

        # Truncate the messages to fit the model's context window
        truncated_messages = truncate_messages(
            messages, self.config.get("model", self.default_model)
        )

        # Query the LLM for preprocessing suggestions
        preprocessing_prompt = (
            "Based on the following reduce operation prompt, should we
do any preprocessing on the input data? "
            "Consider if we need to remove unnecessary context, or
logically construct an output that will help in the task. "
            "If preprocessing would be beneficial, explain why and
suggest specific steps. If not, explain why preprocessing isn't
necessary.\n\n"
            f"Reduce operation prompt:\n{truncated_messages[0]
['content']}"
        )

        preprocessing_response = self.llm_client.generate_rewrite(
            model=self.config.get("model", self.default_model),
            messages=[{"role": "user", "content":
preprocessing_prompt}],
            response_format={
                "type": "json_object",
                "schema": {
                    "type": "object",
                    "properties": {
                        "preprocessing_needed": {"type": "boolean"},
                        "rationale": {"type": "string"},
                        "suggested_steps": {"type": "string"},
                    },
                    "required": [
                        "preprocessing_needed",
                        "rationale",
                        "suggested_steps",
                    ],
                },
            },
        )

        preprocessing_result =
preprocessing_response.choices[0].message.content

        should_preprocess = preprocessing_result["preprocessing_needed"]
        preprocessing_rationale = preprocessing_result["rationale"]

        self.console.log("[bold]Map-Reduce Decomposition Analysis:
[/bold]")
        self.console.log(f"Should write a map operation:
```

```
{should_preprocess}")
        self.console.log(f"Rationale: {preprocessing_rationale}")

        if should_preprocess:
            self.console.log(
                f"Suggested steps:
{preprocessing_result['suggested_steps']}"
            )

        return should_preprocess,
preprocessing_result["suggested_steps"]

    def _optimize_single_reduce(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        validator_prompt: str,
    ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
        """
        Optimize a single reduce operation.

        This method performs the following steps:
        1. Determine and configure value sampling
        2. Determine if the reduce operation is associative
        3. Create and evaluate multiple reduce plans
        4. Run the best reduce plan

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.
            validator_prompt (str): The validator prompt for evaluating
reduce plans.

        Returns:
            tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A
tuple containing a single-item list with the optimized configuration
            and a single-item list with the output from the optimized
operation, and the cost of the operation due to synthesizing any resolve
operations.
        """
        # Step 1: Determine and configure value sampling (TODO: re-
enable this when the agent is more reliable)
        # value_sampling_config =
self._determine_value_sampling(op_config, input_data)
        # if value_sampling_config["enabled"]:
        #     op_config["value_sampling"] = value_sampling_config
        #     self.console.log("[bold]Value Sampling Configuration:
[/bold]")
        #     self.console.log(json.dumps(value_sampling_config,
indent=2))

        # Step 2: Determine if the reduce operation is associative
        is_associative = self._is_associative(op_config, input_data)

        # Step 3: Create and evaluate multiple reduce plans
        self.console.post_optimizer_status(StageType.CANDIDATE_PLANS)
        self.console.log("[bold magenta]Generating batched plans...
[/bold magenta]")
        reduce_plans = self._create_reduce_plans(op_config, input_data,
is_associative)

        # Create gleaning plans
        self.console.log("[bold magenta]Generating gleaning plans...
[/bold magenta]")
        gleaning_plans = self._generate_gleaning_plans(reduce_plans,
validator_prompt)

        self.console.log("[bold magenta]Evaluating plans...[/bold
magenta]")
        self.console.post_optimizer_status(StageType.EVALUATION_RESULTS)
        best_plan = self._evaluate_reduce_plans(
            op_config, reduce_plans + gleaning_plans, input_data,
validator_prompt
        )

        # Step 4: Run the best reduce plan
        optimized_output = self._run_operation(best_plan, input_data)
        self.console.post_optimizer_status(StageType.END)

        return [best_plan], optimized_output, 0.0

    def _generate_gleaning_plans(
        self,
        plans: list[dict[str, Any]],
        validation_prompt: str,
    ) -> list[dict[str, Any]]:
        """
        Generate plans that use gleaning for the given operation.

        Gleaning involves iteratively refining the output of an
operation
        based on validation feedback. This method creates plans with
different
        numbers of gleaning rounds.

        Args:
            plans (list[dict[str, Any]]): The list of plans to use for
gleaning.
            validation_prompt (str): The prompt used for validating the
operation's output.

        Returns:
            dict[str, list[dict[str, Any]]]: A dictionary of gleaning
plans, where each key
            is a plan name and each value is a list containing a single
operation configuration
            with gleaning parameters.

        """
        # Generate an op with gleaning num_rounds and validation_prompt
        gleaning_plans = []
        gleaning_rounds = [1]
        biggest_batch_size = max([plan["fold_batch_size"] for plan in
plans])
        for plan in plans:
            if plan["fold_batch_size"] != biggest_batch_size:
                continue
            for gleaning_round in gleaning_rounds:
                plan_copy = copy.deepcopy(plan)
                plan_copy["gleaning"] = {
                    "num_rounds": gleaning_round,
                    "validation_prompt": validation_prompt,
                }
                plan_name =
f"gleaning_{gleaning_round}_rounds_{plan['name']}"
                plan_copy["name"] = plan_name
                gleaning_plans.append(plan_copy)
        return gleaning_plans

    def _optimize_decomposed_reduce(
        self,
        decomposition_result: dict[str, Any],
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        level: int,
    ) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
        """
        Optimize a decomposed reduce operation.

        This method performs the following steps:
        1. Group the input data by the sub-group key.
        2. Optimize the first reduce operation.
        3. Run the optimized first reduce operation on all groups.
        4. Optimize the second reduce operation using the results of the
first.
        5. Run the optimized second reduce operation.

        Args:
            decomposition_result (dict[str, Any]): The result of the
decomposition evaluation.
            op_config (dict[str, Any]): The original reduce operation
configuration.
            input_data (list[dict[str, Any]]): The input data for the
reduce operation.
            level (int): The current level of decomposition.
        Returns:
            tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A
tuple containing the list of optimized configurations
            for both reduce operations and the final output of the
second reduce operation, and the cost of the operation due to
synthesizing any resolve operations.
        """
        sub_group_key = decomposition_result["sub_group_key"]
        first_reduce_prompt =
decomposition_result["first_reduce_prompt"]
        second_reduce_prompt =
decomposition_result["second_reduce_prompt"]
        pipeline = []
        all_cost = 0.0

        first_reduce_config = op_config.copy()
        first_reduce_config["prompt"] = first_reduce_prompt
        if isinstance(op_config["reduce_key"], list):
            first_reduce_config["reduce_key"] = [sub_group_key] +
op_config[
                "reduce_key"
            ]
        else:
            first_reduce_config["reduce_key"] = [sub_group_key,
op_config["reduce_key"]]
        first_reduce_config["pass_through"] = True

        if first_reduce_config.get("synthesize_resolve", True):
            resolve_config = {
                "name": f"synthesized_resolve_{uuid.uuid4().hex[:8]}",
                "type": "resolve",
                "empty": True,
                "embedding_model": "text-embedding-3-small",
                "resolution_model": self.config.get("default_model",
"gpt-4o-mini"),
                "comparison_model": self.config.get("default_model",
"gpt-4o-mini"),
                "_intermediates": {
                    "map_prompt": op_config.get("_intermediates",
{}).get(
                        "last_map_prompt"
                    ),
                    "reduce_key": first_reduce_config["reduce_key"],
                },
            }
            optimized_resolve_config, resolve_cost = JoinOptimizer(
                self.runner,
                self.config,
                resolve_config,
                self.console,
                self.llm_client,
                self.max_threads,
            ).optimize_resolve(input_data)
            all_cost += resolve_cost

            if not optimized_resolve_config.get("empty", False):
                # Add this to the pipeline
                pipeline += [optimized_resolve_config]

                # Run the resolver
                optimized_output = self._run_operation(
                    optimized_resolve_config, input_data
                )
                input_data = optimized_output

        first_optimized_configs, first_outputs, first_cost =
self.optimize(
            first_reduce_config, input_data, level + 1
        )
        pipeline += first_optimized_configs
        all_cost += first_cost

        # Optimize second reduce operation
        second_reduce_config = op_config.copy()
        second_reduce_config["prompt"] = second_reduce_prompt
        second_reduce_config["pass_through"] = True

        second_optimized_configs, second_outputs, second_cost =
self.optimize(
            second_reduce_config, first_outputs, level + 1
        )

        # Combine optimized configs and return with final output
        pipeline += second_optimized_configs
        all_cost += second_cost

        return pipeline, second_outputs, all_cost
```

```

    def _evaluate_decomposition(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        level: int = 1,
    ) -> dict[str, Any]:
        """
        Evaluate whether decomposing the reduce operation would be
beneficial.

        This method first determines if decomposition would be helpful,
and if so,
        it then determines the sub-group key and prompts for the
decomposed operations.

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.
            level (int): The current level of decomposition.

        Returns:
            dict[str, Any]: A dictionary containing the decomposition
decision and details.
        """
        should_decompose = self._should_decompose(op_config, input_data,
level)

        # Log the decomposition decision
        if should_decompose["should_decompose"]:
            self.console.log(
                f"[bold green]Decomposition recommended:[/bold green]
{should_decompose['explanation']}"
            )
        else:
            self.console.log(
                f"[bold yellow]Decomposition not recommended:[/bold
yellow] {should_decompose['explanation']}"
            )

        # Return early if decomposition is not recommended
        if not should_decompose["should_decompose"]:
            return should_decompose

        # Temporarily stop the status
        if self.status:
            self.status.stop()

        # Ask user if they agree with the decomposition assessment
        user_agrees = Confirm.ask(
            f"Do you agree with the decomposition assessment? "
            f"[bold]{'Recommended' if
should_decompose['should_decompose'] else 'Not recommended'}[/bold]",
            console=self.console,
        )

        # If user disagrees, invert the decomposition decision
        if not user_agrees:
            should_decompose["should_decompose"] = not should_decompose[
                "should_decompose"
            ]
            should_decompose["explanation"] = (
                "User disagreed with the initial assessment."
            )

        # Restart the status
        if self.status:
            self.status.start()

        # Return if decomposition is not recommended
        if not should_decompose["should_decompose"]:
            return should_decompose

        decomposition_details =
self._get_decomposition_details(op_config, input_data)
        result = {**should_decompose, **decomposition_details}
        if decomposition_details["sub_group_key"] in
op_config["reduce_key"]:
            result["should_decompose"] = False
            result[
                "explanation"
            ] += " However, the suggested sub-group key is already part
of the current reduce key(s), so decomposition is not recommended."
            result["sub_group_key"] = ""

        return result

    def _should_decompose(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        level: int = 1,
    ) -> dict[str, Any]:
        """
        Determine if decomposing the reduce operation would be
beneficial.

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.
            level (int): The current level of decomposition.

        Returns:
            dict[str, Any]: A dictionary containing the decomposition
decision and explanation.
        """
        # TODO: we have not enabled recursive decomposition yet
        if level > 1 and not op_config.get("recursively_optimize",
False):
            return {
                "should_decompose": False,
                "explanation": "Recursive decomposition is not
enabled.",
            }

        system_prompt = (
            "You are an AI assistant tasked with optimizing data
processing pipelines."
```

```
        )

        # Sample a subset of input data for analysis
        sample_size = min(10, len(input_data))
        sample_input = random.sample(input_data, sample_size)

        # Get all keys from the input data
        all_keys = set().union(*(item.keys() for item in sample_input))
        reduce_key = op_config["reduce_key"]
        reduce_keys = [reduce_key] if isinstance(reduce_key, str) else
reduce_key
        other_keys = [key for key in all_keys if key not in reduce_keys]

        # See if there's an input schema and constrain the sample_input
to that schema
        input_schema = op_config.get("input", {}).get("schema", {})
        if input_schema:
            sample_input = [
                {key: item[key] for key in input_schema} for item in
sample_input
            ]

        # Create a sample of values for other keys
        sample_values = {
            key: list(set(str(item.get(key))[:50] for item in
sample_input))[:5]
            for key in other_keys
        }

        prompt = f"""Analyze the following reduce operation and
determine if it should be decomposed into two reduce operations chained
together:

        Reduce Operation Prompt:
        ```
        {op_config['prompt']}
        ```

        Current Reduce Key(s): {reduce_keys}
        Other Available Keys: {', '.join(other_keys)}

        Sample values for other keys:
        {json.dumps(sample_values, indent=2)}

        Based on this information, determine if it would be beneficial
to decompose this reduce operation into a sub-reduce operation followed
by a final reduce operation. Consider ALL of the following:

        1. Is there a natural hierarchy in the data (e.g., country ->
state -> city) among the other available keys, with a key at a finer
level of granularity than the current reduce key(s)?
        2. Are the current reduce key(s) some form of ID, and are there
many different types of inputs for that ID among the other available
keys?
        3. Does the prompt implicitly ask for sub-grouping based on the
other available keys (e.g., "summarize policies by state, then by
country")?
        4. Would splitting the operation improve accuracy (i.e., make
sure information isn't lost when reducing)?
        5. Are all the keys of the potential hierarchy provided in the
other available keys? If not, we should not decompose.
```

```
        6. Importantly, do not suggest decomposition using any key that
is already part of the current reduce key(s). We are looking for a new
key from the other available keys to use for sub-grouping.
        7. Do not suggest keys that don't contain meaningful information
(e.g., id-related keys).

        Provide your analysis in the following format:
        """

        parameters = {
            "type": "object",
            "properties": {
                "should_decompose": {"type": "boolean"},
                "explanation": {"type": "string"},
            },
            "required": ["should_decompose", "explanation"],
        }

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

    def _get_decomposition_details(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
    ) -> dict[str, Any]:
        """
        Determine the sub-group key and prompts for decomposed reduce
operations.

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.

        Returns:
            dict[str, Any]: A dictionary containing the sub-group key
and prompts for decomposed operations.
        """
        system_prompt = (
            "You are an AI assistant tasked with optimizing data
processing pipelines."
        )

        # Sample a subset of input data for analysis
        sample_size = min(10, len(input_data))
        sample_input = random.sample(input_data, sample_size)

        # Get all keys from the input data
        all_keys = set().union(*(item.keys() for item in sample_input))
        reduce_key = op_config["reduce_key"]
        reduce_keys = [reduce_key] if isinstance(reduce_key, str) else
reduce_key
        other_keys = [key for key in all_keys if key not in reduce_keys]

        prompt = f"""Given that we've decided to decompose the following
```

```
reduce operation, suggest a two-step reduce process:

        Reduce Operation Prompt:
        ```
        {op_config['prompt']}
        ```

        Reduce Key(s): {reduce_key}
        Other Keys: {', '.join(other_keys)}

        Provide the following:
        1. A sub-group key to use for the first reduce operation
        2. A prompt for the first reduce operation
        3. A prompt for the second (final) reduce operation

        For the reduce operation prompts, you should only minimally
modify the original prompt. The prompts should be Jinja templates, and
the only variables they can access are the `reduce_key` and `inputs`
variables.

        Provide your suggestions in the following format:
        """

        parameters = {
            "type": "object",
            "properties": {
                "sub_group_key": {"type": "string"},
                "first_reduce_prompt": {"type": "string"},
                "second_reduce_prompt": {"type": "string"},
            },
            "required": [
                "sub_group_key",
                "first_reduce_prompt",
                "second_reduce_prompt",
            ],
        }

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

    def _determine_value_sampling(
        self, op_config: dict[str, Any], input_data: list[dict[str,
Any]]
    ) -> dict[str, Any]:
        """
        Determine whether value sampling should be enabled and configure
its parameters.
        """
        system_prompt = (
            "You are an AI assistant helping to optimize data processing
pipelines."
        )

        # Sample a subset of input data for analysis
        sample_size = min(100, len(input_data))
        sample_input = random.sample(input_data, sample_size)

```

```
        prompt = f"""
        Analyze the following reduce operation and determine if value
sampling should be enabled:

        Reduce Operation Prompt:
        {op_config['prompt']}

        Sample Input Data (first 2 items):
        {json.dumps(sample_input[:2], indent=2)}

        Value sampling is appropriate for reduce operations that don't
need to look at all the values for each key to produce a good result,
such as generic summarization tasks.

        Based on the reduce operation prompt and the sample input data,
determine if value sampling should be enabled.
        Answer with 'yes' if value sampling should be enabled or 'no' if
it should not be enabled. Explain your reasoning briefly.
        """

        parameters = {
            "type": "object",
            "properties": {
                "enable_sampling": {"type": "boolean"},
                "explanation": {"type": "string"},
            },
            "required": ["enable_sampling", "explanation"],
        }

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

        if not result["enable_sampling"]:
            return {"enabled": False}

        # Print the explanation for enabling value sampling
        self.console.log(f"Value sampling enabled:
{result['explanation']}")

        # Determine sampling method
        prompt = f"""
        We are optimizing a reduce operation in a data processing
pipeline. The reduce operation is defined by the following prompt:

        Reduce Operation Prompt:
        {op_config['prompt']}

        Sample Input Data (first 2 items):
        {json.dumps(sample_input[:2], indent=2)}

        We have determined that value sampling should be enabled for
this reduce operation. Value sampling is a technique used to process
only a subset of the input data for each reduce key, rather than
processing all items. This can significantly reduce processing time and
costs for very large datasets, especially when the reduce operation
doesn't require looking at every single item to produce a good result
(e.g., summarization tasks).
```

```

        Now we need to choose the most appropriate sampling method. The
available methods are:

        1. "random": Randomly select a subset of values.
        Example: In a customer review analysis task, randomly selecting
a subset of reviews to summarize the overall sentiment.

        2. "cluster": Use K-means clustering to select representative
samples.
        Example: In a document categorization task, clustering documents
based on their content and selecting representative documents from each
cluster to determine the overall categories.

        3. "sem_sim": Use semantic similarity to select the most
relevant samples to a query text.
        Example: In a news article summarization task, selecting
articles that are semantically similar to a query like "Major economic
events of {{reduce_key}}" to produce a focused summary.

        Based on the reduce operation prompt, the nature of the task,
and the sample input data, which sampling method would be most
appropriate?

        Provide your answer as either "random", "cluster", or "sem_sim",
and explain your reasoning in detail. Consider the following in your
explanation:
        - The nature of the reduce task (e.g., summarization,
aggregation, analysis)
        - The structure and content of the input data
        - The potential benefits and drawbacks of each sampling method
for this specific task
        """

        parameters = {
            "type": "object",
            "properties": {
                "method": {"type": "string", "enum": ["random",
"cluster", "sem_sim"]},
                "explanation": {"type": "string"},
            },
            "required": ["method", "explanation"],
        }

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

        value_sampling_config = {
            "enabled": True,
            "method": method,
            "sample_size": 100,  # Default sample size
            "embedding_model": "text-embedding-3-small",
        }

        if method in ["cluster", "sem_sim"]:
            # Determine embedding keys
            prompt = f"""
            For the {method} sampling method, we need to determine which
keys from the input data should be used for generating embeddings.

            Input data keys:
            {', '.join(sample_input[0].keys())}

            Sample Input Data:
            {json.dumps(sample_input[0], indent=2)[:1000]}...

            Based on the reduce operation prompt and the sample input
data, which keys should be used for generating embeddings? Use keys that
will create meaningful embeddings (i.e., not id-related keys).
            Provide your answer as a list of key names that is a subset
of the input data keys. You should pick only the 1-3 keys that are
necessary for generating meaningful embeddings, that have relatively
short values.
            """

            parameters = {
                "type": "object",
                "properties": {
                    "embedding_keys": {"type": "array", "items":
{"type": "string"}},
                    "explanation": {"type": "string"},
                },
                "required": ["embedding_keys", "explanation"],
            }

            response = self.llm_client.generate_rewrite(
                [{"role": "user", "content": prompt}],
                system_prompt,
                parameters,
            )
            result = json.loads(response.choices[0].message.content)
            # TODO: validate that these exist
            embedding_keys = result["embedding_keys"]
            for key in result["embedding_keys"]:
                if key not in sample_input[0]:
                    embedding_keys.remove(key)

            if not embedding_keys:
                # Select the reduce key
                self.console.log(
                    "No embedding keys found, selecting reduce key for
embedding key"
                )
                embedding_keys = (
                    op_config["reduce_key"]
                    if isinstance(op_config["reduce_key"], list)
                    else [op_config["reduce_key"]]
                )

            value_sampling_config["embedding_keys"] = embedding_keys

        if method == "sem_sim":
            # Determine query text
            prompt = f"""
            For the semantic similarity (sem_sim) sampling method, we
need to determine the query text to compare against when selecting
samples.
```

```

            Reduce Operation Prompt:
            {op_config['prompt']}

            The query text should be a Jinja template with access to the
`reduce_key` variable.
            Based on the reduce operation prompt, what would be an
appropriate query text for selecting relevant samples?
            """

            parameters = {
                "type": "object",
                "properties": {
                    "query_text": {"type": "string"},
                    "explanation": {"type": "string"},
                },
                "required": ["query_text", "explanation"],
            }

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

        return value_sampling_config

    def _is_associative(
        self, op_config: dict[str, Any], input_data: list[dict[str,
Any]]
    ) -> bool:
        """
        Determine if the reduce operation is associative.

        This method analyzes the reduce operation configuration and a
sample of the input data
        to determine if the operation is associative (i.e., the order of
combining elements
        doesn't affect the final result).

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.

        Returns:
            bool: True if the operation is determined to be associative,
False otherwise.
        """
        system_prompt = (
            "You are an AI assistant helping to optimize data processing
pipelines."
        )

        # Sample a subset of input data for analysis
        sample_size = min(5, len(input_data))
        sample_input = random.sample(input_data, sample_size)

        prompt = f"""
        Analyze the following reduce operation and determine if it is
associative:

        Reduce Operation Prompt:
        {op_config['prompt']}

        Sample Input Data:
        {json.dumps(sample_input, indent=2)[:1000]}...

        Based on the reduce operation prompt, determine whether the
order in which we process data matters.
        Answer with 'yes' if order matters or 'no' if order doesn't
matter.
        Explain your reasoning briefly.

        For example:
        - Merging extracted key-value pairs from documents does not
require order: combining {{"name": "John", "age": 30}} with {{"city":
"New York", "job": "Engineer"}} yields the same result regardless of
order
        - Generating a timeline of events requires order: the order of
events matters for maintaining chronological accuracy.

        Consider these examples when determining whether the order in
which we process data matters. You might also have to consider the
specific data.
        """

        parameters = {
            "type": "object",
            "properties": {
                "order_matters": {"type": "boolean"},
                "explanation": {"type": "string"},
            },
            "required": ["order_matters", "explanation"],
        }

        response = self.llm_client.generate_rewrite(
            [{"role": "user", "content": prompt}],
            system_prompt,
            parameters,
        )
        result = json.loads(response.choices[0].message.content)
        result["is_associative"] = not result["order_matters"]

        self.console.log(
            f"[yellow]Reduce operation {'is associative' if
result['is_associative'] else 'is not associative'}.[/yellow] Analysis:
{result['explanation']}"
        )
        return result["is_associative"]

    def _generate_validator_prompt(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        original_output: list[dict[str, Any]],
    ) -> str:
        """
        Generate a custom validator prompt for assessing the quality of
```

```
the reduce operation output.

        This method creates a prompt that will be used to validate the
output of the reduce operation.
        It includes specific questions about the quality and
completeness of the output.

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.
            original_output (list[dict[str, Any]]): Original output of
the reduce operation.

        Returns:
            str: A custom validator prompt as a string.
        """
        system_prompt = "You are an AI assistant tasked with creating
custom validation prompts for reduce operations in data processing
pipelines."

        sample_input = random.choice(input_data)
        input_keys = op_config.get("input", {}).get("schema", {})
        if input_keys:
            sample_input = {k: sample_input[k] for k in input_keys}

        reduce_key = op_config.get("reduce_key")
        if reduce_key and original_output:
            if isinstance(reduce_key, list):
                key = next(
                    (
                        tuple(item[k] for k in reduce_key)
                        for item in original_output
                        if all(k in item for k in reduce_key)
                    ),
                    tuple(None for _ in reduce_key),
                )
                sample_output = next(
                    (
                        item
                        for item in original_output
                        if all(item.get(k) == v for k, v in
zip(reduce_key, key))
                    ),
                    {},
                )
            else:
                key = next(
                    (
                        item[reduce_key]
                        for item in original_output
                        if reduce_key in item
                    ),
                    None,
                )
                sample_output = next(
                    (item for item in original_output if
item.get(reduce_key) == key),
                    {},
                )
        else:
            sample_output = original_output[0] if original_output else
{}

        output_keys = op_config.get("output", {}).get("schema", {})
        sample_output = {k: sample_output[k] for k in output_keys}

        prompt = f"""
        Analyze the following reduce operation and its input/output:

        Reduce Operation Prompt:
        {op_config["prompt"]}

        Sample Input (just one item):
        {json.dumps(sample_input, indent=2)}

        Sample Output:
        {json.dumps(sample_output, indent=2)}

        Create a custom validator prompt that will assess how well the
reduce operation performed its intended task. The prompt should ask
specific 2-3 questions about the quality of the output, such as:
        1. Does the output accurately reflect the aggregation method
specified in the task? For example, if finding anomalies, are the
identified anomalies actually anomalies?
        2. Are there any missing fields, unexpected null values, or data
type mismatches in the output compared to the expected schema?
        3. Does the output maintain the key information from the input
while appropriately condensing or summarizing it? For instance, in a
text summarization task, are the main points preserved?
        4. How well does the output adhere to any specific formatting
requirements mentioned in the original prompt, such as character limits
for summaries or specific data types for aggregated values?

        Note that the output may reflect more than just the input
provided, since we only provide a one-item sample input. Provide your
response as a single string containing the custom validator prompt. The
prompt should be tailored to the task and avoid generic criteria. The
prompt should not reference a specific value in the sample input, but
rather a general property.

        Your prompt should not have any placeholders like {{ reduce_key
}} or {{ input_key }}. It should just be a string.
        """

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

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

    def _validate_reduce_output(
        self,
        op_config: dict[str, Any],
        validation_inputs: dict[Any, list[dict[str, Any]]],
        output_data: list[dict[str, Any]],
        validator_prompt: str,
    ) -> dict[str, Any]:
        """
        Validate the output of the reduce operation using the generated
validator prompt.

        This method assesses the quality of the reduce operation output
by applying the validator prompt
        to multiple samples of the input and output data.

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            validation_inputs (dict[Any, list[dict[str, Any]]]):
Validation inputs for the reduce operation.
            output_data (list[dict[str, Any]]): Output data from the
reduce operation.
            validator_prompt (str): The validator prompt generated
earlier.

        Returns:
            dict[str, Any]: A dictionary containing validation results
and a flag indicating if improvement is needed.
        """
        system_prompt = "You are an AI assistant tasked with validating
the output of reduce operations in data processing pipelines."

        validation_results = []
        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            futures = []
            for reduce_key, inputs in validation_inputs.items():
                if (
                    op_config["reduce_key"] == ["_all"]
                    or op_config["reduce_key"] == "_all"
                ):
                    sample_output = output_data[0]
                elif isinstance(op_config["reduce_key"], list):
                    sample_output = next(
                        (
                            item
                            for item in output_data
                            if all(
                                item[key] == reduce_key[i]
                                for i, key in
enumerate(op_config["reduce_key"])
                            )
                        ),
                        None,
                    )
                else:
                    sample_output = next(
                        (
                            item
                            for item in output_data
                            if item[op_config["reduce_key"]] ==
reduce_key
                        ),
                        None,
                    )

                if sample_output is None:
                    self.console.log(
                        f"Warning: No output found for reduce key
{reduce_key}"
                    )
                    continue

                input_str = json.dumps(inputs, indent=2)
                # truncate input_str to 40,000 words
                input_str = input_str.split()[:40000]
                input_str = " ".join(input_str) + "..."

                prompt = f"""{validator_prompt}

                Reduce Operation Task:
                {op_config["prompt"]}

                Input Data Samples:
                {input_str}

                Output Data Sample:
                {json.dumps(sample_output, indent=2)}

                Based on the validator prompt and the input/output
samples, assess the quality (e.g., correctness, completeness) of the
reduce operation output.
                Provide your assessment in the following format:
                """

                parameters = {
                    "type": "object",
                    "properties": {
                        "is_correct": {"type": "boolean"},
                        "issues": {"type": "array", "items": {"type":
"string"}},
                        "suggestions": {"type": "array", "items":
{"type": "string"}},
                    },
                    "required": ["is_correct", "issues", "suggestions"],
                }

                futures.append(
                    executor.submit(
                        self.llm_client.generate_judge,
                        [{"role": "user", "content": prompt}],
                        system_prompt,
                        parameters,
                    )
                )

            for future, (reduce_key, inputs) in zip(futures,
validation_inputs.items()):
                response = future.result()
                result = json.loads(response.choices[0].message.content)
                validation_results.append(result)

        # Determine if optimization is needed based on validation
results
```

```
        invalid_count = sum(
            1 for result in validation_results if not
result["is_correct"]
        )
        needs_improvement = invalid_count > 1 or (
            invalid_count == 1 and len(validation_results) == 1
        )

        return {
            "needs_improvement": needs_improvement,
            "validation_results": validation_results,
        }

    def _create_validation_inputs(
        self, input_data: list[dict[str, Any]], reduce_key: str |
list[str]
    ) -> dict[Any, list[dict[str, Any]]]:
        # Group input data by reduce_key
        grouped_data = {}
        if reduce_key == ["_all"]:
            # Put all data in one group under a single key
            grouped_data[("_all",)] = input_data
        else:
            # Group by reduce key(s) as before
            for item in input_data:
                if isinstance(reduce_key, list):
                    key = tuple(item[k] for k in reduce_key)
                else:
                    key = item[reduce_key]
                if key not in grouped_data:
                    grouped_data[key] = []
                grouped_data[key].append(item)

        # Select a fixed number of reduce keys
        selected_keys = random.sample(
            list(grouped_data.keys()),
            min(self.num_samples_in_validation, len(grouped_data)),
        )

        # Create a new dict with only the selected keys
        validation_inputs = {key: grouped_data[key] for key in
selected_keys}

        return validation_inputs

    def _create_reduce_plans(
        self,
        op_config: dict[str, Any],
        input_data: list[dict[str, Any]],
        is_associative: bool,
    ) -> list[dict[str, Any]]:
        """
        Create multiple reduce plans based on the input data and
operation configuration.

        This method generates various reduce plans by varying batch
sizes and fold prompts.
        It takes into account the LLM's context window size to determine
appropriate batch sizes.

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            input_data (list[dict[str, Any]]): Input data for the reduce
operation.
            is_associative (bool): Flag indicating whether the reduce
operation is associative.

        Returns:
            list[dict[str, Any]]: A list of reduce plans, each with
different batch sizes and fold prompts.
        """
        model = op_config.get("model", "gpt-4o-mini")
        model_input_context_length = model_cost.get(model, {}).get(
            "max_input_tokens", 8192
        )

        # Estimate tokens for prompt, input, and output
        prompt_tokens = count_tokens(op_config["prompt"], model)
        sample_input = input_data[:100]
        sample_output = self._run_operation(op_config, input_data[:100])

        prompt_vars = extract_jinja_variables(op_config["prompt"])
        prompt_vars = [var.split(".")[-1] for var in prompt_vars]
        avg_input_tokens = mean(
            [
                count_tokens(
                    json.dumps({k: item[k] for k in prompt_vars if k in
item}), model
                )
                for item in sample_input
            ]
        )
        avg_output_tokens = mean(
            [
                count_tokens(
                    json.dumps({k: item[k] for k in prompt_vars if k in
item}), model
                )
                for item in sample_output
            ]
        )

        # Calculate max batch size that fits in context window
        max_batch_size = (
            model_input_context_length - prompt_tokens -
avg_output_tokens
        ) // avg_input_tokens

        # Generate 6 candidate batch sizes
        batch_sizes = [
            max(1, int(max_batch_size * ratio))
            for ratio in [0.1, 0.2, 0.4, 0.6, 0.75, 0.9]
        ]
        # Log the generated batch sizes
        self.console.log("[cyan]Generating plans for batch sizes:
[/cyan]")
        for size in batch_sizes:
            self.console.log(f"  - {size}")
        batch_sizes = sorted(set(batch_sizes))  # Remove duplicates and
sort

```

```
        plans = []

        # Generate multiple fold prompts
        max_retries = 5
        retry_count = 0
        fold_prompts = []

        while retry_count < max_retries and not fold_prompts:
            try:
                fold_prompts = self._synthesize_fold_prompts(
                    op_config,
                    sample_input,
                    sample_output,
                    num_prompts=self.num_fold_prompts,
                )
                fold_prompts = list(set(fold_prompts))
                if not fold_prompts:
                    raise ValueError("No fold prompts generated")
            except Exception as e:
                retry_count += 1
                if retry_count == max_retries:
                    raise RuntimeError(
                        f"Failed to generate fold prompts after
{max_retries} attempts: {str(e)}"
                    )
                self.console.log(
                    f"Retry {retry_count}/{max_retries}: Failed to
generate fold prompts. Retrying..."
                )

        for batch_size in batch_sizes:
            for fold_idx, fold_prompt in enumerate(fold_prompts):
                plan = op_config.copy()
                plan["fold_prompt"] = fold_prompt
                plan["fold_batch_size"] = batch_size
                plan["associative"] = is_associative
                plan["name"] = f"
{op_config['name']}_bs_{batch_size}_fp_{fold_idx}"
                plans.append(plan)

        return plans

    def _calculate_compression_ratio(
        self,
        op_config: dict[str, Any],
        sample_input: list[dict[str, Any]],
        sample_output: list[dict[str, Any]],
    ) -> float:
        """
        Calculate the compression ratio of the reduce operation.

        This method compares the size of the input data to the size of
the output data
        to determine how much the data is being compressed by the reduce
operation.

        Args:
            op_config (dict[str, Any]): Configuration for the reduce
operation.
            sample_input (list[dict[str, Any]]): Sample input data.
            sample_output (list[dict[str, Any]]): Sample output data.

        Returns:
            float: The calculated compression ratio.
        """
        reduce_key = op_config["reduce_key"]
        input_schema = op_config.get("input", {}).get("schema", {})
        output_schema = op_config["output"]["schema"]
        model = op_config.get("model", "gpt-4o-mini")

        compression_ratios = {}

        # Handle both single key and list of keys
        if isinstance(reduce_key, list):
            distinct_keys = set(
                tuple(item[k] for k in reduce_key) for item in
sample_input
            )
        else:
            distinct_keys = set(item[reduce_key] for item in
sample_input)

        for key in distinct_keys:
            if isinstance(reduce_key, list):
                key_input = [
                    item
                    for item in sample_input
                    if tuple(item[k] for k in reduce_key) == key
                ]
                key_output = [
                    item
                    for item in sample_output
                    if tuple(item[k] for k in reduce_key) == key
                ]
            else:
                key_input = [item for item in sample_input if
item[reduce_key] == key]
                key_output = [item for item in sample_output if
item[reduce_key] == key]

            if input_schema:
                key_input_tokens = sum(
                    count_tokens(
                        json.dumps({k: item[k] for k in input_schema if
k in item}),
                        model,
                    )
                    for item in key_input
                )
            else:
                key_input_tokens = sum(
                    count_tokens(json.dumps(item), model) for item in
key_input
                )

            key_output_tokens = sum(
                count_tokens(
                    json.dumps({k: item[k] for k in output_schema if k
in item}), model
                )
                for item in key_output
            )

            compression_ratios[key] = (
                key_output_tokens / key_input_tokens if key_input_tokens
> 0 else 1
            )

        if not compression_ratios:
            return 1

        # Calculate importance weights based on the number of items for
each key
        total_items = len(sample_input)
        if isinstance(reduce_key, list):
            importance_weights = {
                key: len(
                    [
                        item
                        for item in sample_input
                        if tuple(item[k] for k in reduce_key) == key
                    ]
                )
                / total_items
                for key in compression_ratios
            }
        else:
            importance_weights = {
                key: len([item for item in sample_input if
item[reduce_key] == key])
                / total_items
                for key in compression_ratios
            }

        # Calculate weighted average of compression ratios
        weighted_sum = sum(
            compression_ratios[key] * importance_weights[key]
            for key in compression_ratios
        )
        return weighted_sum

    def _synthesize_fold_prompts(
        self,
        op_config: dict[str, Any],
        sample_input: list[dict[str, Any]],
        sample_output: list[dict[str, Any]],
        num_prompts: int = 2,
    ) -> list[str]:
        """
        Synthesize fold prompts for the reduce operation. We generate
multiple
        fold prompts in case one is bad.

        A fold operation is a higher-order function that iterates
through a data structure,
        accumulating the results of applying a given combining operation
to its elements.
        In the context of reduce operations, folding allows processing
of data in batches,
        which can significantly improve performance for large datasets.

        This method generates multiple fold prompts that can be used to
optimize the reduce operation
```

```
        by allowing it to run on batches of inputs. It uses the language
model to create prompts
        that are variations of the original reduce prompt, adapted for
folding operations.

        Args:
            op_config (dict[str, Any]): The configuration of the reduce
operation.
            sample_input (list[dict[str, Any]]): A sample of the input
data.
            sample_output (list[dict[str, Any]]): A sample of the output
data.
            num_prompts (int, optional): The number of fold prompts to
generate. Defaults to 2.

        Returns:
            list[str]: A list of synthesized fold prompts.

        The method performs the following steps:
        1. Sets up the system prompt and parameters for the language
model.
        2. Defines a function to get random examples from the sample
data.
        3. Creates a prompt template for generating fold prompts.
        4. Uses multi-threading to generate multiple fold prompts in
parallel.
        5. Returns the list of generated fold prompts.
        """
        system_prompt = "You are an AI assistant tasked with creating a
fold prompt for reduce operations in data processing pipelines."
        original_prompt = op_config["prompt"]

        input_schema = op_config.get("input", {}).get("schema", {})
        output_schema = op_config["output"]["schema"]

        def get_random_examples():
            reduce_key = op_config["reduce_key"]
            reduce_key = (
                list(reduce_key) if not isinstance(reduce_key, list)
else reduce_key
            )

            if reduce_key == ["_all"]:
                # For _all case, just pick random input and output
examples
                input_example = random.choice(sample_input)
                output_example = random.choice(sample_output)
            elif isinstance(reduce_key, list):
                random_key = tuple(
                    random.choice(
                        [
                            tuple(item[k] for k in reduce_key if k in
item)
                            for item in sample_input
                            if all(k in item for k in reduce_key)
                        ]
                    )
                )
                input_example = random.choice(
                    [
                        item
                        for item in sample_input
                        if all(item.get(k) == v for k, v in
zip(reduce_key, random_key))
                    ]
                )
                output_example = random.choice(
                    [
                        item
                        for item in sample_output
                        if all(item.get(k) == v for k, v in
zip(reduce_key, random_key))
                    ]
                )

            if input_schema:
                input_example = {
                    k: input_example[k] for k in input_schema if k in
input_example
                }
            output_example = {
                k: output_example[k] for k in output_schema if k in
output_example
            }
            return input_example, output_example

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

        def generate_single_prompt():
            input_example, output_example = get_random_examples()
            prompt = f"""
            Original Reduce Operation Prompt:
            {original_prompt}

            Sample Input:
            {json.dumps(input_example, indent=2)}

            Sample Output:
            {json.dumps(output_example, indent=2)}

            Create a fold prompt for the reduce operation to run on
batches of inputs. The fold prompt should:
            1. Minimally modify the original reduce prompt
            2. Describe how to combine the new values with the current
reduced value
            3. Be designed to work iteratively, allowing for multiple
fold operations. The first iteration will use the original prompt, and
all successive iterations will use the fold prompt.

            The fold prompt should be a Jinja2 template with the
following variables available:
            - {{{{ output }}}}: The current reduced value (a dictionary
with the current output schema)
            - {{{{ inputs }}}}: A list of new values to be folded in
            - {{{{ reduce_key }}}}: The key used for grouping in the
reduce operation

            Provide the fold prompt as a string.
            """
            response = self.llm_client.generate_rewrite(
                [{"role": "user", "content": prompt}],
                system_prompt,
                parameters,
            )
            fold_prompt =
json.loads(response.choices[0].message.content)["fold_prompt"]

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

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

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

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

{original_prompt}

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

{{{{ output }}}}

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

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

        return fold_prompts
```

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

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

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

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

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

        The method performs the following steps:
        1. Evaluates each plan using multi-threading.
        2. Sorts the plans based on their evaluation scores.
        3. Selects the best plan and attempts to create a merged plan.
        4. Evaluates the merged plan and compares it to the best
original plan.
        5. Returns either the merged plan or the best original plan
based on their scores.
        """
        self.console.log("\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
`config`	`dict[str, Any]`	Configuration dictionary for the optimizer.	*required*
`console`	`Console`	Rich console object for pretty printing.	*required*
`llm_client`	`LLMClient`	Client for interacting with a language model.	*required*

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

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

```python
def __init__(
    self,
    runner,
    run_operation: Callable,
    num_fold_prompts: int = 1,
    num_samples_in_validation: int = 10,
):
    """
    Initialize the ReduceOptimizer.

    Args:
        config (dict[str, Any]): Configuration dictionary for the
optimizer.
        console (Console): Rich console object for pretty printing.
        llm_client (LLMClient): Client for interacting with a language
model.
        max_threads (int): Maximum number of threads to use for parallel
processing.
        run_operation (Callable): Function to run an operation.
        num_fold_prompts (int, optional): Number of fold prompts to
generate. Defaults to 1.
        num_samples_in_validation (int, optional): Number of samples to
use in validation. Defaults to 10.
    """
    self.runner = runner
    self.config = self.runner.config
    self.console = self.runner.console
    self.llm_client = self.runner.optimizer.llm_client
    self._run_operation = run_operation
    self.max_threads = self.runner.max_threads
    self.num_fold_prompts = num_fold_prompts
    self.num_samples_in_validation = num_samples_in_validation
    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
`op_config`	`dict[str, Any]`	Configuration for the reduce operation.	*required*
`input_data`	`list[dict[str, Any]]`	Input data for the reduce operation.	*required*

Returns:

Type	Description
`list[dict[str, Any]]`	tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A tuple containing the list of optimized configurations
`list[dict[str, Any]]`	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`

```
def optimize(
    self,
    op_config: dict[str, Any],
    input_data: list[dict[str, Any]],
    level: int = 1,
) -> tuple[list[dict[str, Any]], list[dict[str, Any]], float]:
    """
    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)

    Args:
        op_config (dict[str, Any]): Configuration for the reduce
operation.
        input_data (list[dict[str, Any]]): Input data for the reduce
operation.

    Returns:
        tuple[list[dict[str, Any]], list[dict[str, Any]], float]: A tuple
containing the list of optimized configurations
        and the list of outputs from the optimized operation(s), and the
cost of the operation due to synthesizing any resolve operations.
    """
    (
        validation_results,
        prompt_tokens,
        model_input_context_length,
        model,
        validator_prompt,
        original_output,
    ) = self.should_optimize_helper(op_config, input_data)

    # add_map_op = False
    if prompt_tokens * 2 > model_input_context_length:
        # add_map_op = True
        self.console.log(
            f"[yellow]Warning: The reduce prompt exceeds the token limit
for model {model}. "
            f"Token count: {prompt_tokens}, Limit:
{model_input_context_length}. "
            f"Add a map operation to the pipeline.[/yellow]"
        )

    # # Also query an agent to look at a sample of the inputs and see if
they think a map operation would be helpful
    # preprocessing_steps = ""
    # should_use_map, preprocessing_steps = self._should_use_map(
    #     op_config, input_data
    # )
    # if should_use_map or add_map_op:
    #     # Synthesize a map operation
    #     map_prompt, map_output_schema = self._synthesize_map_operation(
    #         op_config, preprocessing_steps, input_data
    #     )
    #     # Change the reduce operation prompt to use the map schema
    #     new_reduce_prompt =
self._change_reduce_prompt_to_use_map_schema(
    #         op_config["prompt"], map_output_schema
    #     )
    #     op_config["prompt"] = new_reduce_prompt

    #     # Return unoptimized map and reduce operations
    #     return [map_prompt, op_config], input_data, 0.0

    # Print the validation results
    self.console.log("[bold]Validation Results on Initial Sample:
[/bold]")
    if validation_results["needs_improvement"] or self.config.get(
        "optimizer_config", {}
    ).get("force_decompose", False):
        self.console.post_optimizer_rationale(
            should_optimize=True,
            rationale="\n".join(
                [
                    f"Issues: {result['issues']} Suggestions:
{result['suggestions']}"
                    for result in
validation_results["validation_results"]
                ]
            ),
            validator_prompt=validator_prompt,
        )
        self.console.log(
            "\n".join(
                [
                    f"Issues: {result['issues']} Suggestions:
{result['suggestions']}"
                    for result in
validation_results["validation_results"]
                ]
            )
        )

        # Step 3: Evaluate if decomposition is beneficial
        decomposition_result = self._evaluate_decomposition(
            op_config, input_data, level
        )

        if decomposition_result["should_decompose"]:
            return self._optimize_decomposed_reduce(
                decomposition_result, op_config, input_data, level
            )

        return self._optimize_single_reduce(op_config, input_data,
validator_prompt)
    else:
        self.console.log(f"No improvements identified;
{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`

```
class JoinOptimizer:
    def __init__(
        self,
        runner,
        op_config: dict[str, Any],
        target_recall: float = 0.95,
        sample_size: int = 500,
        sampling_weight: float = 20,
        agent_max_retries: int = 5,
        estimated_selectivity: float | None = None,
    ):
        self.runner = runner
        self.config = runner.config
        self.op_config = op_config
        self.llm_client = runner.optimizer.llm_client
        self.max_threads = runner.max_threads
        self.console = runner.console
        self.target_recall = target_recall
        self.sample_size = sample_size
        self.sampling_weight = sampling_weight
        self.agent_max_retries = agent_max_retries
        self.estimated_selectivity = estimated_selectivity
        self.console.log(f"Target Recall: {self.target_recall}")
        self.status = self.runner.status
        self.max_comparison_sampling_attempts = 5
        self.synthesized_keys = []
        # if self.estimated_selectivity is not None:
        #     self.console.log(
        #         f"[yellow]Using estimated selectivity of
{self.estimated_selectivity}[/yellow]"
        #     )

    def _analyze_map_prompt_categorization(self, map_prompt: str) ->
tuple[bool, str]:
        """
        Analyze the map prompt to determine if it's explicitly
categorical.

        Args:
            map_prompt (str): The map prompt to analyze.

        Returns:
            bool: True if the prompt is explicitly categorical, False
otherwise.
        """
        messages = [
            {
                "role": "system",
                "content": "You are an AI assistant tasked with
analyzing prompts for data processing operations.",
            },
            {
                "role": "user",
                "content": f"""Analyze the following map operation
prompt and determine if it is explicitly categorical,
                meaning it details a specific set of possible outputs:

```

```
                {map_prompt}

                Respond with 'Yes' if the prompt is explicitly
categorical, detailing a finite set of possible outputs.
                Respond with 'No' if the prompt allows for open-ended or
non-categorical responses.
                Provide a brief explanation for your decision.""",
            },
        ]

        response = self.llm_client.generate_rewrite(
            messages,
            "You are an expert in analyzing natural language prompts for
data processing tasks.",
            {
                "type": "object",
                "properties": {
                    "is_categorical": {
                        "type": "string",
                        "enum": ["Yes", "No"],
                        "description": "Whether the prompt is explicitly
categorical",
                    },
                    "explanation": {
                        "type": "string",
                        "description": "Brief explanation for the
decision",
                    },
                },
                "required": ["is_categorical", "explanation"],
            },
        )

        analysis = json.loads(response.choices[0].message.content)

        self.console.log("[bold]Map Prompt Analysis:[/bold]")
        self.console.log(f"Is Categorical:
{analysis['is_categorical']}")
        self.console.log(f"Explanation: {analysis['explanation']}")

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

    def _determine_duplicate_keys(
        self,
        input_data: list[dict[str, Any]],
        reduce_key: list[str],
        map_prompt: str | None = None,
    ) -> tuple[bool, str]:
        # Prepare a sample of the input data for analysis
        sample_size = min(10, len(input_data))
        data_sample = random.sample(
            [{rk: item[rk] for rk in reduce_key} for item in
input_data], sample_size
        )

        context_prefix = ""
        if map_prompt:
            context_prefix = f"For context, these values came out of a
pipeline with the following prompt:\n\n{map_prompt}\n\n"

```

```
        messages = [
            {
                "role": "user",
                "content": f"{context_prefix}I want to do a reduce
operation on these values, and I need to determine if there are semantic
duplicates in the data, where the strings are different but they
technically belong in the same group. Note that exact string duplicates
should not be considered here.\n\nHere's a sample of the data (showing
the '{reduce_key}' field(s)): {data_sample}\n\nBased on this {'context
and ' if map_prompt else ''}sample, are there likely to be such semantic
duplicates (not exact string matches) in the dataset? Respond with 'yes'
only if you think there are semantic duplicates, or 'no' if you don't
see evidence of semantic duplicates or if you only see exact string
duplicates.",
            },
        ]
        response = self.llm_client.generate_rewrite(
            messages,
            "You are an expert data analyst. Analyze the given data
sample and determine if there are likely to be semantic duplicate values
that belong in the same group, even if the strings are different.",
            {
                "type": "object",
                "properties": {
                    "likely_duplicates": {
                        "type": "string",
                        "enum": ["Yes", "No"],
                        "description": "Whether duplicates are likely to
exist in the full dataset",
                    },
                    "explanation": {
                        "type": "string",
                        "description": "Brief explanation for the
decision",
                    },
                },
                "required": ["likely_duplicates", "explanation"],
            },
        )

        analysis = json.loads(response.choices[0].message.content)

        self.console.log(f"[bold]Duplicate Analysis for '{reduce_key}':
[/bold]")
        self.console.log(f"Likely Duplicates:
{analysis['likely_duplicates']}")
        self.console.log(f"Explanation: {analysis['explanation']}")

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

    def _sample_random_pairs(
        self, input_data: list[dict[str, Any]], n: int
    ) -> list[tuple[int, int]]:
        """Sample random pairs of indices, excluding exact matches."""
        pairs = set()
        max_attempts = n * 10  # Avoid infinite loop
        attempts = 0

        while len(pairs) < n and attempts < max_attempts:
            i, j = random.sample(range(len(input_data)), 2)
            if i != j and input_data[i] != input_data[j]:
                pairs.add((min(i, j), max(i, j)))  # Ensure ordered pairs
            attempts += 1

        return list(pairs)

    def _check_duplicates_with_llm(
        self,
        input_data: list[dict[str, Any]],
        pairs: list[tuple[int, int]],
        reduce_key: list[str],
        map_prompt: str | None = None,
    ) -> tuple[bool, str]:
        """Use LLM to check if any pairs are duplicates."""

        content = "Analyze the following pairs of entries and determine if any of them are likely duplicates. Respond with 'Yes' if you find any likely duplicates, or 'No' if none of the pairs seem to be duplicates. Provide a brief explanation for your decision.\n\n"

        if map_prompt:
            content = (
                f"For reference, here is the map prompt used earlier in the pipeline: {map_prompt}\n\n"
                + content
            )

        for i, (idx1, idx2) in enumerate(pairs, 1):
            content += f"Pair {i}:\n"
            content += "Entry 1:\n"
            for key in reduce_key:
                content += f"{key}: {json.dumps(input_data[idx1][key], indent=2)}\n"
            content += "\nEntry 2:\n"
            for key in reduce_key:
                content += f"{key}: {json.dumps(input_data[idx2][key], indent=2)}\n"
            content += "\n"

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

        system_prompt = "You are an AI assistant tasked with identifying potential duplicate entries in a dataset."
        response_schema = {
            "type": "object",
            "properties": {
                "duplicates_found": {"type": "string", "enum": ["Yes", "No"]},
                "explanation": {"type": "string"},
            },
            "required": ["duplicates_found", "explanation"],
        }

        response = self.llm_client.generate_rewrite(
            messages, system_prompt, response_schema
        )

        # Print the duplicates_found and explanation
        self.console.log(
            f"[bold]Duplicates in keys found:[/bold]
{response['duplicates_found']}\n"
            f"[bold]Explanation:[/bold] {response['explanation']}"
        )

        return response["duplicates_found"].lower() == "yes",
response["explanation"]

    def synthesize_compare_prompt(
        self, map_prompt: str | None, reduce_key: list[str]
    ) -> str:

        system_prompt = f"You are an AI assistant tasked with creating a
comparison prompt for LLM-assisted entity resolution. Your task is to
create a comparison prompt that will be used to compare two entities,
referred to as input1 and input2, to see if they are likely the same
entity based on the following reduce key(s): {', '.join(reduce_key)}."
        if map_prompt:
            system_prompt += f"\n\nFor context, here is the prompt used
earlier in the pipeline to create the inputs to resolve: {map_prompt}"

        messages = [
            {
                "role": "user",
                "content": f"""
    Create a comparison prompt for entity resolution: The prompt should:
    1. Be tailored to the specific domain and type of data being
compared ({reduce_key}), based on the context provided.
    2. Instruct to compare two entities, referred to as input1 and
input2.
    3. Specifically mention comparing each reduce key in input1 and
input2 (e.g., input1.{{key}} and input2.{{key}} for each key in
{reduce_key}). You can reference other fields in the input as well, as
long as they are short.
    4. Include instructions to consider relevant attributes or
characteristics for comparison.
    5. Ask to respond with "True" if the entities are likely the same,
or "False" if they are likely different.

    Example structure:
    ```
    Compare the following two {reduce_key} from [entity or document
type]:

    [Entity 1]:
    {{{{ input1.key1 }}}}
    {{{{ input1.optional_key2 }}}}

    [Entity 2]:
    {{{{ input2.key1 }}}}
    {{{{ input2.optional_key2 }}}}

    Are these [entities] likely referring to the same [entity type]?
Consider [list relevant attributes or characteristics to compare].
Respond with "True" if they are likely the same [entity type], or
"False" if they are likely different [entity types].
    ```
```

```

    Please generate the comparison prompt, which should be a Jinja2 template:
    """,
            }
        ]

        response = self.llm_client.generate_rewrite(
            messages,
            system_prompt,
            {
                "type": "object",
                "properties": {
                    "comparison_prompt": {
                        "type": "string",
                        "description": "Detailed comparison prompt for entity resolution",
                    }
                },
                "required": ["comparison_prompt"],
            },
        )

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

        # Log the synthesized comparison prompt
        self.console.log("[green]Synthesized comparison prompt:[/green]")
        self.console.log(comparison_prompt)

        if not comparison_prompt:
            raise ValueError(
                "Could not synthesize a comparison prompt. Please provide a comparison prompt in the config."
            )

        return comparison_prompt

    def synthesize_resolution_prompt(
        self,
        map_prompt: str | None,
        reduce_key: list[str],
        output_schema: dict[str, str],
    ) -> str:
        system_prompt = f"""You are an AI assistant tasked with creating a resolution prompt for LLM-assisted entity resolution.
        Your task is to create a prompt that will be used to merge multiple duplicate keys into a single, consolidated key.
        The key(s) being resolved (known as the reduce_key) are {', '.join(reduce_key)}.
        The duplicate keys will be provided in a list called 'inputs' in a Jinja2 template.
        """

        if map_prompt:
            system_prompt += f"\n\nFor context, here is the prompt used earlier in the pipeline to create the inputs to resolve: {map_prompt}"

        messages = [
            {
                "role": "user",
                "content": f"""
    Create a resolution prompt for merging duplicate keys into a single
key. The prompt should:
    1. Be tailored to the specific domain and type of data being merged,
based on the context provided.
    2. Use a Jinja2 template to iterate over the duplicate keys
(accessed as 'inputs', where each item is a dictionary containing the
reduce_key fields, which you can access as entry.reduce_key for each
reduce_key in {reduce_key}).
    3. Instruct to create a single, consolidated key from the duplicate
keys.
    4. Include guidelines for resolving conflicts (e.g., choosing the
most recent, most complete, or most reliable information).
    5. Specify that the output of the resolution prompt should conform
to the given output schema: {json.dumps(output_schema, indent=2)}

    Example structure:
    ```
    Analyze the following duplicate entries for the {reduce_key} key:

    {{% for key in inputs %}}
    Entry {{{{ loop.index }}}}:
    {{ % for key in reduce_key %}}
    {{{{ key }}}}: {{{{ key[reduce_key] }}}}
    {{% endfor %}}

    {{% endfor %}}

    Merge these into a single key.
    When merging, follow these guidelines:
    1. [Provide specific merging instructions relevant to the data type]
    2. [Do not make the prompt too long]

    Ensure that the merged key conforms to the following schema:
    {json.dumps(output_schema, indent=2)}

    Return the consolidated key as a single [appropriate data type]
value.
    ```

    Please generate the resolution prompt:
    """,
            }
        ]

        response = self.llm_client.generate_rewrite(
            messages,
            system_prompt,
            {
                "type": "object",
                "properties": {
                    "resolution_prompt": {
                        "type": "string",
                        "description": "Detailed resolution prompt for
merging duplicate keys",
                    }
                },
                "required": ["resolution_prompt"],
```

```
            },
        )

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

        # Log the synthesized resolution prompt
        self.console.log("[green]Synthesized resolution prompt:
[/green]")
        self.console.log(resolution_prompt)

        if not resolution_prompt:
            raise ValueError(
                "Could not synthesize a resolution prompt. Please
provide a resolution prompt in the config."
            )

        return resolution_prompt

    def should_optimize(self, input_data: list[dict[str, Any]]) ->
tuple[bool, str]:
        """
        Determine if the given operation configuration should be
optimized.
        """
        # If there are no blocking keys or embeddings, then we don't
need to optimize
        if not self.op_config.get("blocking_conditions") or not
self.op_config.get(
            "blocking_threshold"
        ):
            return True, ""

        # Check if the operation is marked as empty
        elif self.op_config.get("empty", False):
            # Extract the map prompt from the intermediates
            map_prompt = self.op_config["_intermediates"]["map_prompt"]
            reduce_key = self.op_config["_intermediates"]["reduce_key"]

            if reduce_key is None:
                raise ValueError(
                    "[yellow]Warning: No reduce key found in
intermediates for synthesized resolve operation.[/yellow]"
                )

            dedup = True
            explanation = "There is a reduce operation that does not
follow a resolve operation. Consider adding a resolve operation to
deduplicate the data."

            if map_prompt:
                # Analyze the map prompt
                analysis, explanation =
self._analyze_map_prompt_categorization(
                    map_prompt
                )

                if analysis:
                    dedup = False
            else:
                self.console.log(
                    "[yellow]No map prompt found in intermediates for analysis.[/yellow]"
                )

            # TODO: figure out why this would ever be the case
            if not map_prompt:
                map_prompt = "N/A"

            if dedup is False:
                dedup, explanation = self._determine_duplicate_keys(
                    input_data, reduce_key, map_prompt
                )

            # Now do the last attempt of pairwise comparisons
            if dedup is False:
                # Sample up to 20 random pairs of keys for duplicate analysis
                sampled_pairs = self._sample_random_pairs(input_data, 20)

                # Use LLM to check for duplicates
                duplicates_found, explanation = self._check_duplicates_with_llm(
                    input_data, sampled_pairs, reduce_key, map_prompt
                )

                if duplicates_found:
                    dedup = True

            return dedup, explanation

        return False, ""

    def optimize_resolve(
        self, input_data: list[dict[str, Any]]
    ) -> tuple[dict[str, Any], float]:
        # Check if the operation is marked as empty
        if self.op_config.get("empty", False):
            # Extract the map prompt from the intermediates
            dedup, _ = self.should_optimize(input_data)
            reduce_key = self.op_config["_intermediates"]["reduce_key"]
            map_prompt = self.op_config["_intermediates"]["map_prompt"]

            if dedup is False:
                # If no deduplication is needed, return the same config with 0 cost
                return self.op_config, 0.0

            # Add the reduce key to the output schema in the config
            self.op_config["output"] = {"schema": {rk: "string" for rk in reduce_key}}
            for attempt in range(2):  # Try up to 2 times
                self.op_config["comparison_prompt"] = self.synthesize_compare_prompt(
                    map_prompt, reduce_key
                )
                if (
                    "input1" in self.op_config["comparison_prompt"]
                    and "input2" in self.op_config["comparison_prompt"]
                ):
                    break
                elif attempt == 0:
                    self.console.log(
                        "[yellow]Warning: 'input1' or 'input2' not found
in comparison prompt. Retrying...[/yellow]"
                    )
            if (
                "input1" not in self.op_config["comparison_prompt"]
                or "input2" not in self.op_config["comparison_prompt"]
            ):
                self.console.log(
                    "[red]Error: Failed to generate comparison prompt
with 'input1' and 'input2'. Using last generated prompt.[/red]"
                )
            for attempt in range(2):  # Try up to 2 times
                self.op_config["resolution_prompt"] =
self.synthesize_resolution_prompt(
                    map_prompt, reduce_key, self.op_config["output"]
["schema"]
                )
                if "inputs" in self.op_config["resolution_prompt"]:
                    break
                elif attempt == 0:
                    self.console.log(
                        "[yellow]Warning: 'inputs' not found in
resolution prompt. Retrying...[/yellow]"
                    )
            if "inputs" not in self.op_config["resolution_prompt"]:
                self.console.log(
                    "[red]Error: Failed to generate resolution prompt
with 'inputs'. Using last generated prompt.[/red]"
                )

            # Pop off the empty flag
            self.op_config.pop("empty")

        embeddings, blocking_keys, embedding_cost =
self._compute_embeddings(input_data)
        self.console.log(
            f"[bold]Cost of creating embeddings on the sample:
${embedding_cost:.4f}[/bold]"
        )

        similarities = self._calculate_cosine_similarities(embeddings)

        sampled_pairs = self._sample_pairs(similarities)
        comparison_results, comparison_cost =
self._perform_comparisons_resolve(
            input_data, sampled_pairs
        )

        self._print_similarity_histogram(similarities,
comparison_results)

        threshold, estimated_selectivity = self._find_optimal_threshold(
            comparison_results, similarities
        )

        blocking_rules = self._generate_blocking_rules(
            blocking_keys, input_data, comparison_results
        )

        if blocking_rules:
            false_negatives, rule_selectivity =
self._verify_blocking_rule(
                input_data,
                blocking_rules[0],
                blocking_keys,
                comparison_results,
            )
            # If more than 50% of the sample is false negatives, reject
the blocking rule
            if len(false_negatives) > len(sampled_pairs) / 2:
                if false_negatives:
                    self.console.log(
                        f"[red]Blocking rule rejected.
{len(false_negatives)} false negatives detected in the sample
({len(false_negatives) / len(sampled_pairs):.2f} of the sample).[/red]"
                    )
                    for i, j in false_negatives[:5]:  # Show up to 5
examples
                        self.console.log(
                            f"  Filtered pair: {{ {blocking_keys[0]}:
{input_data[i][blocking_keys[0]]} }} and {{ {blocking_keys[0]}:
{input_data[j][blocking_keys[0]]} }}"
                        )
                    if len(false_negatives) > 5:
                        self.console.log(f"  ... and
{len(false_negatives) - 5} more.")
                blocking_rules = (
                    []
                )  # Clear the blocking rule if it introduces false
negatives or is too selective
            elif not false_negatives and rule_selectivity >
estimated_selectivity:
                self.console.log(
                    "[green]Blocking rule verified. No false negatives
detected in the sample and selectivity is within estimated selectivity.
[/green]"
                )
            else:
                # TODO: ask user if they want to use the blocking rule,
or come up with some good default behavior
                blocking_rules = []

        optimized_config = self._update_config(threshold, blocking_keys,
blocking_rules)
        return optimized_config, embedding_cost + comparison_cost

    def optimize_equijoin(
        self,
        left_data: list[dict[str, Any]],
        right_data: list[dict[str, Any]],
        skip_map_gen: bool = False,
        skip_containment_gen: bool = False,
    ) -> tuple[dict[str, Any], float, dict[str, Any]]:
        left_keys = self.op_config.get("blocking_keys", {}).get("left",
[])
        right_keys = self.op_config.get("blocking_keys",
{}).get("right", [])

```

```
        if not left_keys and not right_keys:
            # Ask the LLM agent if it would be beneficial to do a map operation on
            # one of the datasets before doing an equijoin
            apply_transformation, dataset_to_transform, reason = (
                (
                    self._should_apply_map_transformation(
                        left_keys, right_keys, left_data, right_data
                    )
                )
                if not skip_map_gen
                else (False, None, None)
            )

            if apply_transformation and not skip_map_gen:
                self.console.log(
                    f"LLM agent suggested applying a map transformation to {dataset_to_transform} dataset because: {reason}"
                )
                extraction_prompt, output_key, new_comparison_prompt = (
                    self._generate_map_and_new_join_transformation(
                        dataset_to_transform, reason, left_data, right_data
                    )
                )
                self.console.log(
                    f"Generated map transformation prompt: {extraction_prompt}"
                )
                self.console.log(f"\nNew output key: {output_key}")
                self.console.log(
                    f"\nNew equijoin comparison prompt: {new_comparison_prompt}"
                )

                # Update the comparison prompt
                self.op_config["comparison_prompt"] = new_comparison_prompt

                # Add the output key to the left_keys or right_keys
                if dataset_to_transform == "left":
                    left_keys.append(output_key)
                else:
                    right_keys.append(output_key)

                # Reset the blocking keys in the config
                self.op_config["blocking_keys"] = {
                    "left": left_keys,
                    "right": right_keys,
                }

                # Bubble up this config and return the transformation prompt, so we can optimize the map operation
                return (
                    self.op_config,
                    0.0,
                    {
                        "optimize_map": True,
                        "map_prompt": extraction_prompt,
                        "output_key": output_key,
                        "dataset_to_transform": dataset_to_transform,
                    },
                )

            # Print the reason for not applying a map transformation
            self.console.log(
                f"Reason for not synthesizing a map transformation for
either left or right dataset: {reason}"
            )

        # If there are no blocking keys, generate them
        if not left_keys or not right_keys:
            generated_left_keys, generated_right_keys = (
                self._generate_blocking_keys_equijoin(left_data,
right_data)
            )
            left_keys.extend(generated_left_keys)
            right_keys.extend(generated_right_keys)
            left_keys = list(set(left_keys))
            right_keys = list(set(right_keys))

            # Log the generated blocking keys
            self.console.log(
                "[bold]Generated blocking keys (for embeddings-based
blocking):[/bold]"
            )
            self.console.log(f"Left keys: {left_keys}")
            self.console.log(f"Right keys: {right_keys}")

        left_embeddings, _, left_embedding_cost =
self._compute_embeddings(
            left_data, keys=left_keys
        )
        right_embeddings, _, right_embedding_cost =
self._compute_embeddings(
            right_data, keys=right_keys
        )
        self.console.log(
            f"[bold]Cost of creating embeddings on the sample:
${left_embedding_cost + right_embedding_cost:.4f}[/bold]"
        )

        similarities = self._calculate_cross_similarities(
            left_embeddings, right_embeddings
        )

        sampled_pairs = self._sample_pairs(similarities)
        comparison_results, comparison_cost =
self._perform_comparisons_equijoin(
            left_data, right_data, sampled_pairs
        )
        self._print_similarity_histogram(similarities,
comparison_results)
        attempts = 0
        while (
            not any(result[2] for result in comparison_results)
            and attempts < self.max_comparison_sampling_attempts
        ):
            self.console.log(
                "[yellow]No matches found in the current sample.
Resampling pairs to compare...[/yellow]"
            )
            sampled_pairs = self._sample_pairs(similarities)
            comparison_results, current_cost =
self._perform_comparisons_equijoin(
                left_data, right_data, sampled_pairs
            )
            comparison_cost += current_cost
            self._print_similarity_histogram(similarities,
comparison_results)
            attempts += 1

        if not any(result[2] for result in comparison_results):
            # If still no matches after max_comparison_sampling_attempts
attempts, use 99th percentile similarity as threshold
            # This is a heuristic to avoid being in an infinite loop
            # TODO: have a better plan for sampling pairs or avoiding
getting into this situation
            self.console.log(
                f"[yellow]No matches found after
{self.max_comparison_sampling_attempts} attempts. Using 99th percentile
similarity as threshold.[/yellow]"
            )
            threshold = np.percentile([sim[2] for sim in similarities],
99)
            # TODO: figure out how to estimate selectivity
            estimated_selectivity = 0.0
            self.estimated_selectivity = estimated_selectivity

        else:
            threshold, estimated_selectivity =
self._find_optimal_threshold(
                comparison_results, similarities
            )
            self.estimated_selectivity = estimated_selectivity

        blocking_rules = self._generate_blocking_rules_equijoin(
            left_keys, right_keys, left_data, right_data,
comparison_results
        )

        if blocking_rules:
            false_negatives, rule_selectivity =
self._verify_blocking_rule_equijoin(
                left_data,
                right_data,
                blocking_rules[0],
                left_keys,
                right_keys,
                comparison_results,
            )
            if not false_negatives and rule_selectivity <=
estimated_selectivity:
                self.console.log(
                    "[green]Blocking rule verified. No false negatives
detected in the sample and selectivity is within bounds.[/green]"
                )
            else:
                if false_negatives:
                    self.console.log(
                        f"[red]Blocking rule rejected.
{len(false_negatives)} false negatives detected in the sample.[/red]"
                    )
                    for i, j in false_negatives[:5]:  # Show up to 5
examples
                        self.console.log(
                            f"  Filtered pair: Left: {{{',
'.join(f'{key}: {left_data[i][key]}' for key in left_keys)}}} and Right:
{{{', '.join(f'{key}: {right_data[j][key]}' for key in right_keys)}}}"
                        )
                    if len(false_negatives) > 5:
                        self.console.log(f"  ... and
{len(false_negatives) - 5} more.")
                if rule_selectivity > estimated_selectivity:
                    self.console.log(
                        f"[red]Blocking rule rejected. Rule selectivity
({rule_selectivity:.4f}) is higher than the estimated selectivity
({estimated_selectivity:.4f}).[/red]"
                    )
                blocking_rules = (
                    []
                )  # Clear the blocking rule if it introduces false
negatives or is too selective

        containment_rules = self._generate_containment_rules_equijoin(
            left_data, right_data
        )
        if not skip_containment_gen:
            self.console.log(
                f"[bold]Generated {len(containment_rules)} containment
rules. Please select which ones to use as blocking conditions:[/bold]"
            )
            selected_containment_rules = []
            for rule in containment_rules:
                self.console.log(f"[green]{rule}[/green]")
                # Temporarily stop the status
                if self.status:
                    self.status.stop()
                # Use Rich's Confirm for input
                if Confirm.ask("Use this rule?", console=self.console):
                    selected_containment_rules.append(rule)
                # Restart the status
                if self.status:
                    self.status.start()
        else:
            # Take first 2
            selected_containment_rules = containment_rules[:2]

        if len(containment_rules) > 0:
            self.console.log(
                f"[bold]Selected {len(selected_containment_rules)}
containment rules for blocking.[/bold]"
            )
        blocking_rules.extend(selected_containment_rules)

        optimized_config = self._update_config_equijoin(
            threshold, left_keys, right_keys, blocking_rules
        )
        return (
            optimized_config,
            left_embedding_cost + right_embedding_cost +
comparison_cost,
            {},
        )
```

```

    def _should_apply_map_transformation(
        self,
        left_keys: list[str],
        right_keys: list[str],
        left_data: list[dict[str, Any]],
        right_data: list[dict[str, Any]],
        sample_size: int = 5,
    ) -> tuple[bool, str, str]:
        # Sample data
        left_sample = random.sample(left_data, min(sample_size,
len(left_data)))
        right_sample = random.sample(right_data, min(sample_size,
len(right_data)))

        # Get keys and their average lengths
        all_left_keys = {
            k: sum(len(str(d[k])) for d in left_sample) /
len(left_sample)
            for k in left_sample[0].keys()
        }
        all_right_keys = {
            k: sum(len(str(d[k])) for d in right_sample) /
len(right_sample)
            for k in right_sample[0].keys()
        }

        messages = [
            {
                "role": "user",
                "content": f"""Analyze the following datasets and
determine if an additional LLM transformation should be applied to
generate a new key-value pair for easier joining:

                Comparison prompt for the join operation:
{self.op_config.get('comparison_prompt', 'No comparison prompt
provided.')}

                Left dataset keys and average lengths:
{json.dumps(all_left_keys, indent=2)}
                Right dataset keys and average lengths:
{json.dumps(all_right_keys, indent=2)}

                Left dataset sample:
                {json.dumps(left_sample, indent=2)}

                Right dataset sample:
                {json.dumps(right_sample, indent=2)}

                Current keys used for embedding-based ranking of likely
matches:
                Left keys: {left_keys}
                Right keys: {right_keys}

                Consider the following:
                1. Are the current keys sufficient for accurate
embedding-based ranking of likely matches? We don't want to use too many
keys, or keys with too much information, as this will dilute the signal
in the embeddings.
                2. Are there any keys particularly long (e.g., full text
fields), containing information that is not relevant for the join
operation? The dataset with the longer keys should be transformed.
                3. Would a summary or extraction of important
information from long key-value pairs be beneficial? If so, the dataset
with the longer keys should be transformed.
                4. Is there a mismatch in information representation
between the datasets?
                5. Could an additional LLM-generated field improve the
accuracy of embeddings or join comparisons?

                If you believe an additional LLM transformation would be
beneficial, specify which dataset (left or right) should be transformed
and explain why. Otherwise, indicate that no additional transformation
is needed and explain why the current blocking keys are sufficient.""",
            }
        ]

        response = self.llm_client.generate_rewrite(
            messages,
            "You are an AI expert in data analysis and entity
matching.",
            {
                "type": "object",
                "properties": {
                    "apply_transformation": {"type": "boolean"},
                    "dataset_to_transform": {
                        "type": "string",
                        "enum": ["left", "right", "none"],
                    },
                    "reason": {"type": "string"},
                },
                "required": ["apply_transformation",
"dataset_to_transform", "reason"],
            },
        )

        result = json.loads(response.choices[0].message.content)

        return (
            result["apply_transformation"],
            result["dataset_to_transform"],
            result["reason"],
        )

    def _generate_map_and_new_join_transformation(
        self,
        dataset_to_transform: str,
        reason: str,
        left_data: list[dict[str, Any]],
        right_data: list[dict[str, Any]],
        sample_size: int = 5,
    ) -> tuple[str, str, str]:
        # Sample data
        left_sample = random.sample(left_data, min(sample_size,
len(left_data)))
        right_sample = random.sample(right_data, min(sample_size,
len(right_data)))

        target_data = left_sample if dataset_to_transform == "left" else
right_sample

        messages = [
            {
                "role": "user",
                "content": f"""Generate an LLM prompt to transform the
{dataset_to_transform} dataset for easier joining. The transformation
should create a new key-value pair.

                Current comparison prompt for the join operation:
{self.op_config.get('comparison_prompt', 'No comparison prompt
provided.')}

                Target ({dataset_to_transform}) dataset sample:
                {json.dumps(target_data, indent=2)}

                Other ({'left' if dataset_to_transform == "right" else
"right"}) dataset sample:
                {json.dumps(right_sample if dataset_to_transform ==
"left" else left_sample, indent=2)}

                Reason for transforming {dataset_to_transform} dataset:
{reason}

                Please provide:
                1. An LLM prompt to extract a smaller representation of
what is relevant to the join task. The prompt should be a Jinja2
template, referring to any fields in the input data as {{{{
input.field_name }}}}. The prompt should instruct the LLM to return some
**non-empty** string-valued output. The transformation should be
tailored to the join task if possible, not just a generic summary of the
data.
                2. A name for the new output key that will store the
transformed data.
                3. An edited comparison prompt that leverages the new
attribute created by the transformation. This prompt should be a Jinja2
template, referring to any fields in the input data as {{{{
left.field_name }}}} and {{{{ right.field_name }}}}. The prompt should
be the same as the current comparison prompt, but with a new instruction
that leverages the new attribute created by the transformation (in
addition to the other fields in the prompt). The prompt should instruct
the LLM to return a boolean-valued output, like the current comparison
prompt.""",
            }
        ]

        response = self.llm_client.generate_rewrite(
            messages,
            "You are an AI expert in data analysis and decomposing
complex data processing pipelines.",
            {
                "type": "object",
                "properties": {
                    "extraction_prompt": {"type": "string"},
                    "output_key": {"type": "string"},
                    "new_comparison_prompt": {"type": "string"},
                },
                "required": [
                    "extraction_prompt",
                    "output_key",
                    "new_comparison_prompt",
                ],
            },
        )

        result = json.loads(response.choices[0].message.content)

        return (
            result["extraction_prompt"]
            .replace("left.", "input.")
            .replace("right.", "input."),
            result["output_key"],
            result["new_comparison_prompt"],
        )

    def _generate_blocking_keys_equijoin(
        self,
        left_data: list[dict[str, Any]],
        right_data: list[dict[str, Any]],
        sample_size: int = 5,
    ) -> tuple[list[str], list[str]]:
        # Sample data
        left_sample = random.sample(left_data, min(sample_size,
len(left_data)))
        right_sample = random.sample(right_data, min(sample_size,
len(right_data)))

        # Prepare sample data for LLM
        left_keys = list(left_sample[0].keys())
        right_keys = list(right_sample[0].keys())

        messages = [
            {
                "role": "user",
                "content": f"""Given the following sample data from two
datasets, select appropriate blocking keys for an equijoin operation.
                The blocking process works as follows:
                1. We create embeddings for the selected keys from both
datasets.
                2. We use cosine similarity between these embeddings to
filter pairs for more detailed LLM comparison.
                3. Pairs with high similarity will be passed to the LLM
for final comparison.

                The blocking keys should have relatively short values
and be useful for generating embeddings that capture the essence of
potential matches.

                Left dataset keys: {left_keys}
                Right dataset keys: {right_keys}

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

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

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

                Please select one or more keys from each dataset that
would be suitable for blocking. The keys should contain information
that's likely to be similar in matching records and align with the
comparison prompt's focus.""",
            }
        ]

        response = self.llm_client.generate_rewrite(
            messages,
            "You are an expert in entity matching and database
operations.",
            {
                "type": "object",
                "properties": {
                    "left_blocking_keys": {
                        "type": "array",
                        "items": {"type": "string"},
                        "description": "List of selected blocking keys
from the left dataset",
                    },
                    "right_blocking_keys": {
                        "type": "array",
                        "items": {"type": "string"},
                        "description": "List of selected blocking keys
from the right dataset",
                    },
                },
                "required": ["left_blocking_keys",
"right_blocking_keys"],
            },
        )

        result = json.loads(response.choices[0].message.content)
        left_blocking_keys = result["left_blocking_keys"]
        right_blocking_keys = result["right_blocking_keys"]

        return left_blocking_keys, right_blocking_keys

    def _compute_embeddings(
        self,
        input_data: list[dict[str, Any]],
        keys: list[str] | None = None,
        is_join: bool = True,
    ) -> tuple[list[list[float]], list[str], float]:
        if keys is None:
            keys = self.op_config.get("blocking_keys", [])
            if not keys:
                prompt_template =
self.op_config.get("comparison_prompt", "")
                prompt_vars = extract_jinja_variables(prompt_template)
                # Get rid of input, input1, input2
                prompt_vars = [
                    var
                    for var in prompt_vars
                    if var not in ["input", "input1", "input2"]
                ]

                # strip all things before . in the prompt_vars
                keys += list(set([var.split(".")[-1] for var in
prompt_vars]))
            if not keys:
                self.console.log(
                    "[yellow]Warning: No blocking keys found. Using all
keys for blocking.[/yellow]"
```

```
                )
                keys = list(input_data[0].keys())

        model_input_context_length = model_cost.get(
            self.op_config.get("embedding_model", "text-embedding-3-small"), {}
        ).get("max_input_tokens", 8192)
        texts = [
            " ".join(str(item[key]) for key in keys if key in item)[
                :model_input_context_length
            ]
            for item in input_data
        ]

        embeddings = []
        total_cost = 0
        batch_size = 2000
        for i in range(0, len(texts), batch_size):
            batch = texts[i : i + batch_size]
            self.console.log(
                f"[cyan]Processing batch {i//batch_size + 1} of {len(texts)//batch_size + 1}[/cyan]"
            )
            response = self.runner.api.gen_embedding(
                model=self.op_config.get("embedding_model", "text-embedding-3-small"),
                input=batch,
            )
            embeddings.extend([data["embedding"] for data in response["data"]])
            total_cost += completion_cost(response)
        embeddings = [data["embedding"] for data in response["data"]]
        cost = completion_cost(response)
        return embeddings, keys, cost

    def _calculate_cosine_similarities(
        self, embeddings: list[list[float]]
    ) -> list[tuple[int, int, float]]:
        embeddings_array = np.array(embeddings)
        norms = np.linalg.norm(embeddings_array, axis=1)
        dot_products = np.dot(embeddings_array, embeddings_array.T)
        similarities_matrix = dot_products / np.outer(norms, norms)
        i, j = np.triu_indices(len(embeddings), k=1)
        similarities = list(
            zip(i.tolist(), j.tolist(), similarities_matrix[i, j].tolist())
        )
        return similarities

    def _print_similarity_histogram(
        self,
        similarities: list[tuple[int, int, float]],
        comparison_results: list[tuple[int, int, bool]],
    ):
        flat_similarities = [sim[-1] for sim in similarities if sim[-1] != 1]
        hist, bin_edges = np.histogram(flat_similarities, bins=20)
        max_bar_width, max_count = 50, max(hist)
        normalized_hist = [int(count / max_count * max_bar_width) for count in hist]

```

```python
        # Create a dictionary to store true labels
        true_labels = {(i, j): is_match for i, j, is_match in comparison_results}

        self.console.log("\n[bold]Embedding Cosine Similarity Distribution:[/bold]")
        for i, count in enumerate(normalized_hist):
            bar = "█" * count
            label = f"{bin_edges[i]:.2f}-{bin_edges[i+1]:.2f}"

            # Count true matches and not matches in this bin
            true_matches = 0
            not_matches = 0
            labeled_count = 0
            for sim in similarities:
                if bin_edges[i] <= sim[2] < bin_edges[i + 1]:
                    if (sim[0], sim[1]) in true_labels:
                        labeled_count += 1
                        if true_labels[(sim[0], sim[1])]:
                            true_matches += 1
                        else:
                            not_matches += 1

            # Calculate percentages of labeled pairs
            if labeled_count > 0:
                true_match_percent = (true_matches / labeled_count) * 100
                not_match_percent = (not_matches / labeled_count) * 100
            else:
                true_match_percent = 0
                not_match_percent = 0

            self.console.log(
                f"{label}: {bar} "
                f"(Labeled: {labeled_count}/{hist[i]}, [green]{true_match_percent:.1f}% match[/green], [red]{not_match_percent:.1f}% not match[/red])"
            )
        self.console.log("\n")

    def _sample_pairs(
        self, similarities: list[tuple[int, int, float]]
    ) -> list[tuple[int, int]]:
        # Sort similarities in descending order
        sorted_similarities = sorted(similarities, key=lambda x: x[2], reverse=True)

        # Calculate weights using exponential weighting with self.sampling_weight
        similarities_array = np.array([sim[2] for sim in sorted_similarities])
        weights = np.exp(self.sampling_weight * similarities_array)
        weights /= weights.sum()  # Normalize weights to sum to 1

        # Sample pairs based on the calculated weights
        sampled_indices = np.random.choice(
            len(sorted_similarities),
            size=min(self.sample_size, len(sorted_similarities)),
            replace=False,
            p=weights,
        )

        sampled_pairs = [
            (sorted_similarities[i][0], sorted_similarities[i][1])
            for i in sampled_indices
        ]
        return sampled_pairs

    def _calculate_cross_similarities(
        self, left_embeddings: list[list[float]], right_embeddings:
list[list[float]]
    ) -> list[tuple[int, int, float]]:
        left_array = np.array(left_embeddings)
        right_array = np.array(right_embeddings)
        dot_product = np.dot(left_array, right_array.T)
        norm_left = np.linalg.norm(left_array, axis=1)
        norm_right = np.linalg.norm(right_array, axis=1)
        similarities = dot_product / np.outer(norm_left, norm_right)
        return [
            (i, j, sim)
            for i, row in enumerate(similarities)
            for j, sim in enumerate(row)
        ]

    def _perform_comparisons_resolve(
        self, input_data: list[dict[str, Any]], pairs: list[tuple[int,
int]]
    ) -> tuple[list[tuple[int, int, bool]], float]:
        comparisons, total_cost = [], 0
        op = ResolveOperation(
            self.runner,
            self.op_config,
            self.runner.default_model,
            self.max_threads,
            self.console,
            self.status,
        )
        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            futures = [
                executor.submit(
                    op.compare_pair,
                    self.op_config["comparison_prompt"],
                    self.op_config.get(
                        "comparison_model", self.config.get("model",
"gpt-4o-mini")
                    ),
                    input_data[i],
                    input_data[j],
                )
                for i, j in pairs
            ]
            for future, (i, j) in zip(futures, pairs):
                is_match, cost, _ = future.result()
                comparisons.append((i, j, is_match))
                total_cost += cost

        self.console.log(
            f"[bold]Cost of pairwise comparisons on the sample:
${total_cost:.4f}[/bold]"
        )
        return comparisons, total_cost

    def _perform_comparisons_equijoin(
        self,
        left_data: list[dict[str, Any]],
        right_data: list[dict[str, Any]],
        pairs: list[tuple[int, int]],
    ) -> tuple[list[tuple[int, int, bool]], float]:
        comparisons, total_cost = [], 0
        op = EquijoinOperation(
            self.runner,
            self.op_config,
            self.runner.default_model,
            self.max_threads,
            self.console,
            self.status,
        )
        with ThreadPoolExecutor(max_workers=self.max_threads) as
executor:
            futures = [
                executor.submit(
                    op.compare_pair,
                    self.op_config["comparison_prompt"],
                    self.op_config.get(
                        "comparison_model", self.config.get("model",
"gpt-4o-mini")
                    ),
                    left_data[i],
                    right_data[j] if right_data else left_data[j],
                )
                for i, j in pairs
            ]
            for future, (i, j) in zip(futures, pairs):
                is_match, cost = future.result()
                comparisons.append((i, j, is_match))
                total_cost += cost

        self.console.log(
            f"[bold]Cost of pairwise comparisons on the sample:
${total_cost:.4f}[/bold]"
        )
        return comparisons, total_cost

    def _find_optimal_threshold(
        self,
        comparisons: list[tuple[int, int, bool]],
        similarities: list[tuple[int, int, float]],
    ) -> tuple[float, float, float]:
        true_labels = np.array([comp[2] for comp in comparisons])
        sim_dict = {(i, j): sim for i, j, sim in similarities}
        sim_scores = np.array([sim_dict[(i, j)] for i, j, _ in
comparisons])

        thresholds = np.linspace(0, 1, 100)
        precisions, recalls = [], []

        for threshold in thresholds:
            predictions = sim_scores >= threshold
            tp = np.sum(predictions & true_labels)
            fp = np.sum(predictions & ~true_labels)
            fn = np.sum(~predictions & true_labels)

            precision = tp / (tp + fp) if (tp + fp) > 0 else 0
            recall = tp / (tp + fn) if (tp + fn) > 0 else 0

            precisions.append(precision)
            recalls.append(recall)

        valid_indices = [i for i, r in enumerate(recalls) if r >=
self.target_recall]
        if not valid_indices:
            optimal_threshold = float(thresholds[np.argmax(recalls)])
        else:
            optimal_threshold = float(thresholds[max(valid_indices)])

        # Improved selectivity estimation
        all_similarities = np.array([s[2] for s in similarities])
        sampled_similarities = sim_scores

        # Calculate sampling probabilities
        sampling_probs = np.exp(self.sampling_weight *
sampled_similarities)
        sampling_probs /= sampling_probs.sum()

        # Estimate selectivity using importance sampling
        weights = 1 / (len(all_similarities) * sampling_probs)
        numerator = np.sum(weights * true_labels)
        denominator = np.sum(weights)
        selectivity_estimate = numerator / denominator

        self.console.log(
            "[bold cyan]┌─ Estimated Self-Join Selectivity
─────────────────────────┐[/bold cyan]"
        )
        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]└────────────────────────────────────────────────────────────┘[/bold
cyan]"
        )
        self.console.log(
            f"[bold]Chosen similarity threshold for blocking:
{optimal_threshold:.4f}[/bold]"
        )

        return round(optimal_threshold, 4), selectivity_estimate

    def _generate_blocking_rules(
        self,
        blocking_keys: list[str],
        input_data: list[dict[str, Any]],
        comparisons: list[tuple[int, int, bool]],
    ) -> list[str]:
        # 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: input_data[i][key] for key in blocking_keys},
                {key: input_data[j][key] for key in blocking_keys},
                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 entity resolution. This rule will be used in the form:
`eval(blocking_rule, {{"input1": item1, "input2": item2}})`.

    Sample comparisons (note: these are just a few examples and may not
represent all possible cases):
    {json.dumps(sample_datas, indent=2)}

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

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

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

    Where conditions could be comparisons like:
    "input1['field'].lower() == input2['field'].lower()"
    "abs(len(input1['text']) - len(input2['text'])) <= 5"
    "any(word in input1['description'].lower() for word in
input2['description'].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):  # Up to 3 attempts
            # Generate blocking rule using the LLM
            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"],
                },
            )

            # Extract the blocking rule from the LLM response
            blocking_rule = response.choices[0].message.content
            blocking_rule = json.loads(blocking_rule).get("blocking_rule")

            if blocking_rule:
                self.console.log("")  # Print a newline

                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(
                    input_data, blocking_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"Item 1: {json.dumps({key:
input_data[i][key] for key in blocking_keys})}\nItem 2:
{json.dumps({key: input_data[j][key] for key in blocking_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(
        self,
        input_data: list[dict[str, Any]],
        blocking_keys: list[str],
        blocking_rule: str,
        comparisons: list[tuple[int, int, bool]],
    ) -> list[tuple[int, int]]:
        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

        filtered_pairs = []

        for i, j, is_match in comparisons:
            if is_match:
                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 not apply_blocking_rule(item1, item2):
                    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
                self.console.log(
                    f"  Incorrectly filtered pair 1: {json.dumps({key:
input_data[i][key] for key in blocking_keys})}  and pair 2:
{json.dumps({key: input_data[j][key] for key in blocking_keys})}"
                )
            if len(filtered_pairs) > 5:
                self.console.log(
                    f"  ... and {len(filtered_pairs) - 5} more incorrect
pairs."
                )

        return filtered_pairs

    def _generate_containment_rules_equijoin(
        self,
        left_data: list[dict[str, Any]],
        right_data: list[dict[str, Any]],
    ) -> list[str]:
        # Get all available keys from the sample data
        left_keys = set(left_data[0].keys())
        right_keys = set(right_data[0].keys())

        # Find the keys that are in the config's prompt
        try:
            left_prompt_keys = set(
                self.op_config.get("comparison_prompt", "")
                .split("{{ left.")[1]
                .split(" }}")[0]
                .split(".")
            )
        except Exception as e:
            self.console.log(f"[red]Error parsing comparison prompt: {e}
[/red]")
            left_prompt_keys = left_keys

        try:
            right_prompt_keys = set(
                self.op_config.get("comparison_prompt", "")
                .split("{{ right.")[1]
                .split(" }}")[0]
                .split(".")
            )
        except Exception as e:
            self.console.log(f"[red]Error parsing comparison prompt: {e}
[/red]")
            right_prompt_keys = right_keys

        # Sample a few records from each dataset
        sample_left = random.sample(left_data, min(3, len(left_data)))
        sample_right = random.sample(right_data, min(3,
len(right_data)))

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

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

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

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

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

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

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

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

        response = self.llm_client.generate_rewrite(
            messages,
            "You are an expert in data matching and Python
programming.",
            {
                "type": "object",
                "properties": {
                    "containment_rules": {
                        "type": "array",
                        "items": {"type": "string"},
                        "description": "List of containment-based
blocking rules as Python expressions",
                    }
                },
                "required": ["containment_rules"],
            },
        )

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

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

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

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

    Sample comparisons (note: these are just a few examples and may not
represent all possible cases):
    {json.dumps(sample_datas, indent=2)}

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

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

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

    Where conditions could be comparisons like:
    "left['{left_keys[0]}'].lower() == right['{right_keys[0]}'].lower()"
    "abs(len(left['{left_keys[0]}']) - len(right['{right_keys[0]}'])) <=
5"
    "any(word in left['{left_keys[0]}'].lower() for word in
right['{right_keys[0]}'].lower().split())"

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

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

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

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

            if blocking_rule:
                self.console.log("")

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

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

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

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

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

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

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

        filtered_pairs = []

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

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

        return filtered_pairs

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

        false_negatives = []
        total_pairs = 0
        blocked_pairs = 0

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

        rule_selectivity = blocked_pairs / total_pairs if total_pairs >
0 else 0

        return false_negatives, rule_selectivity

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

    def _verify_blocking_rule(
        self,
        input_data: list[dict[str, Any]],
        blocking_rule: str,
        blocking_keys: list[str],
        comparison_results: list[tuple[int, int, bool]],
    ) -> tuple[list[tuple[int, int]], float]:
        def apply_blocking_rule(item1, item2):
            try:
                return eval(blocking_rule, {"input1": item1, "input2":
item2})
            except Exception as e:
                self.console.log(f"[red]Error applying blocking rule:
{e}[/red]")
                return True  # If there's an error, we default to
comparing the pair

        false_negatives = []
        total_pairs = 0
        blocked_pairs = 0

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

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

        rule_selectivity = blocked_pairs / total_pairs if total_pairs >
0 else 0

        return false_negatives, rule_selectivity

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

`should_optimize(input_data)`

Determine if the given operation configuration should be optimized.

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

```
def should_optimize(self, input_data: list[dict[str, Any]]) ->
tuple[bool, str]:
    """
    Determine if the given operation configuration should be optimized.
    """
    # If there are no blocking keys or embeddings, then we don't need to
optimize
    if not self.op_config.get("blocking_conditions") or not
self.op_config.get(
        "blocking_threshold"
    ):
        return True, ""

    # Check if the operation is marked as empty
    elif self.op_config.get("empty", False):
        # Extract the map prompt from the intermediates
        map_prompt = self.op_config["_intermediates"]["map_prompt"]
        reduce_key = self.op_config["_intermediates"]["reduce_key"]

        if reduce_key is None:
            raise ValueError(
                "[yellow]Warning: No reduce key found in intermediates
for synthesized resolve operation.[/yellow]"
            )

        dedup = True
        explanation = "There is a reduce operation that does not follow a
resolve operation. Consider adding a resolve operation to deduplicate the
data."

        if map_prompt:
            # Analyze the map prompt
            analysis, explanation =
self._analyze_map_prompt_categorization(
                map_prompt
            )

            if analysis:
                dedup = False
        else:
            self.console.log(
                "[yellow]No map prompt found in intermediates for
analysis.[/yellow]"
            )

        # TODO: figure out why this would ever be the case
        if not map_prompt:
            map_prompt = "N/A"

        if dedup is False:
            dedup, explanation = self._determine_duplicate_keys(
                input_data, reduce_key, map_prompt
            )

        # Now do the last attempt of pairwise comparisons
        if dedup is False:
            # Sample up to 20 random pairs of keys for duplicate analysis
            sampled_pairs = self._sample_random_pairs(input_data, 20)

            # Use LLM to check for duplicates
            duplicates_found, explanation =
self._check_duplicates_with_llm(
                input_data, sampled_pairs, reduce_key, map_prompt
            )

            if duplicates_found:
                dedup = True

        return dedup, explanation

    return False, ""
```