Parallel Map Operation

The Parallel Map operation in DocETL applies multiple independent transformations to each item in the input data concurrently, maintaining a 1:1 input-to-output ratio while generating multiple fields simultaneously.

Similarity to Map Operation

The Parallel Map operation is very similar to the standard Map operation. The key difference is that Parallel Map allows you to define multiple prompts that run concurrently (without having to explicitly create a DAG), whereas a standard Map operation typically involves a single transformation.

Configuration

Each prompt in the Parallel Map operation is responsible for generating specific fields of the output. The prompts are executed concurrently, improving efficiency when working with multiple transformations.

The output schema should include all the fields generated by the individual prompts, ensuring that the results are combined into a single output item for each input.

Required Parameters

Parameter	Description
`name`	A unique name for the operation
`type`	Must be set to "parallel_map"
`prompts`	A list of prompt configurations (see below)
`output`	Schema definition for the combined output from all prompts

Each prompt configuration in the `prompts` list should contain:

- `prompt`: The prompt template to use for the transformation

- `output_keys` : List of keys that this prompt will generate
- `model` (optional): The language model to use for this specific prompt
- `gleaning` (optional): Advanced validation settings for this prompt (see Per-Prompt Gleaning section below)

Optional Parameters

Parameter	Description	Default
`model`	The default language model to use	Falls back to `default_model`
`optimize`	Flag to enable operation optimization	True
`recursively_optimize`	Flag to enable recursive optimization	false
`sample`	Number of samples to use for the operation	Processes all data
`timeout`	Timeout for each LLM call in seconds	120
`max_retries_per_timeout`	Maximum number of retries per timeout	2
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls.	{}

Why use Parallel Map instead of multiple Map operations?

While you could achieve similar results with multiple Map operations, Parallel Map offers several advantages:

1. **Concurrency**: Prompts run in parallel, potentially reducing overall processing time.
2. **Simplified Configuration**: You define multiple transformations in a single operation, reducing pipeline complexity.
3. **Unified Output**: Results from all prompts are combined into a single output item, simplifying downstream operations.

🚀 Example: Processing Job Applications

Here's an example of a parallel map operation that processes job applications by extracting key information and evaluating candidates:

```yaml
- name: process_job_application
  type: parallel_map
  prompts:
    - name: extract_skills
      prompt: "Given the following resume: '{{ input.resume }}', list the top
5 relevant skills for a software engineering position."
      output_keys:
        - skills
      gleaning:
        num_rounds: 1
        validation_prompt: |
          Confirm the skills list contains **exactly** 5 distinct skills and
each skill is one or two words long.
      model: gpt-4o-mini
    - name: calculate_experience
      prompt: "Based on the work history in this resume: '{{ input.resume }}',
calculate the total years of relevant experience for a software engineering
role."
      output_keys:
        - years_experience
      model: gpt-4o-mini
    - name: evaluate_cultural_fit
      prompt: "Analyze the following cover letter: '{{ input.cover_letter }}'.
Rate the candidate's potential cultural fit on a scale of 1-10, where 10 is
the highest."
      output_keys:
        - cultural_fit_score
      model: gpt-4o-mini
  output:
    schema:
      skills: list[string]
      years_experience: float
      cultural_fit_score: integer
```

This Parallel Map operation processes job applications by concurrently extracting skills, calculating experience, and evaluating cultural fit.

Advanced Validation: Per-Prompt Gleaning

Each prompt in a Parallel Map operation can include its own `gleaning` configuration. Gleaning works exactly as described in the operators overview but is **scoped to the individual LLM call** for that prompt. This allows you to tailor validation logic—and even the model used—to the specific transformation being performed.

The structure of the `gleaning` block is identical:

```yaml
gleaning:
  num_rounds: 1               # maximum refinement iterations
  validation_prompt: |        # judge prompt appended to the chat thread
    Ensure the extracted skills list contains at least 5 distinct items.
  model: gpt-4o-mini          # (optional) model for the validator LLM
```

Example with Per-Prompt Gleaning

```yaml
- name: process_job_application
  type: parallel_map
  prompts:
    - name: extract_skills
      prompt: "Given the following resume: '{{ input.resume }}', list the top
5 relevant skills for a software engineering position."
      output_keys:
        - skills
      gleaning:
        num_rounds: 1
        validation_prompt: |
          Confirm the skills list contains **exactly** 5 distinct skills and
each skill is one or two words long.
      model: gpt-4o-mini
    - name: calculate_experience
      prompt: "Based on the work history in this resume: '{{ input.resume }}',
calculate the total years of relevant experience for a software engineering
role."
      output_keys:
        - years_experience
      gleaning:
        num_rounds: 2
        validation_prompt: |
          Verify that the years of experience is a non-negative number and
round to one decimal place if necessary.
    - name: evaluate_cultural_fit
      prompt: "Analyze the following cover letter: '{{ input.cover_letter }}'.
Rate the candidate's potential cultural fit on a scale of 1-10, where 10 is
the highest."
      output_keys:
        - cultural_fit_score
      model: gpt-4o-mini
  output:
    schema:
      skills: list[string]
      years_experience: float
      cultural_fit_score: integer
```

In this configuration, only the `extract_skills` and `calculate_experience` prompts use gleaning. Each prompt's validator runs **immediately after** its own LLM call and before the overall outputs are merged.

Advantages

1. **Concurrency**: Multiple transformations are applied simultaneously, potentially reducing overall processing time.
2. **Simplicity**: Users can define multiple transformations without needing to create explicit DAGs in the configuration.
3. **Flexibility**: Different models can be used for different prompts within the same operation.
4. **Maintainability**: Each transformation can be defined and updated independently, making it easier to manage complex operations.

Best Practices

1. **Independent Transformations**: Ensure that the prompts in a Parallel Map operation are truly independent of each other to maximize the benefits of concurrent execution.
2. **Balanced Prompts**: Try to design prompts that have similar complexity and execution times to optimize overall performance.
3. **Output Schema Alignment**: Ensure that the output schema correctly captures all the fields generated by the individual prompts.
4. **Lightweight Validators**: When using per-prompt gleaning, keep validation prompts concise so that the cost and latency overhead stays manageable.