Reduce Operation

The Reduce operation in DocETL aggregates data based on a key. It supports both batch reduction and incremental folding for large datasets, making it versatile for various data processing tasks.

Motivation

Reduce operations are essential when you need to summarize or aggregate data across multiple records. For example, you might want to:

- Analyze sentiment trends across social media posts
- Consolidate patient medical records from multiple visits
- Synthesize key findings from a set of research papers on a specific topic

🚀 Example: Summarizing Customer Feedback

Let's look at a practical example of using the Reduce operation to summarize customer feedback by department:

```
- name: summarize_feedback
  type: reduce
  reduce_key: department
  prompt: |
    Summarize the customer feedback for the {{ inputs[0].department }}
department:

    {% for item in inputs %}
    Feedback {{ loop.index }}: {{ item.feedback }}
    {% endfor %}

    Provide a concise summary of the main points and overall sentiment.
  output:
    schema:
      summary: string
      sentiment: string
```

This Reduce operation processes customer feedback grouped by department:

1. Groups all feedback entries by the 'department' key.
2. For each department, it applies the prompt to summarize the feedback and determine overall sentiment.

3. Outputs a summary and sentiment for each department.

Configuration

Required Parameters

- `type`: Must be set to "reduce".
- `reduce_key`: The key (or list of keys) to use for grouping data. Use `_all` to group all data into one group.
- `prompt`: The prompt template to use for the reduction operation.
- `output`: Schema definition for the output from the LLM.

Optional Parameters

Parameter	Description	Default
`sample`	Number of samples to use for the operation	None
`synthesize_resolve`	If false, won't synthesize a resolve operation between map and reduce	true
`model`	The language model to use	Falls back to default_model
`input`	Specifies the schema or keys to subselect from each item	All keys from input items
`pass_through`	If true, non-input keys from the first item in the group will be passed through	false
`associative`	If true, the reduce operation is associative (i.e., order doesn't matter)	true
`fold_prompt`	A prompt template for incremental folding	None
`fold_batch_size`	Number of items to process in each fold operation	None

Parameter	Description	Default
`value_sampling`	A dictionary specifying the sampling strategy for large groups	None
`verbose`	If true, enables detailed logging of the reduce operation	false
`persist_intermediates`	If true, persists the intermediate results for each group to the key `_{operation_name}_intermediates`	false
`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.	{}
`bypass_cache`	If true, bypass the cache for this operation.	False

Advanced Features

Incremental Folding

For large datasets, the Reduce operation supports incremental folding. This allows processing of large groups in smaller batches, which can be more efficient and reduce memory usage.

To enable incremental folding, provide a `fold_prompt` and `fold_batch_size`:

```
- name: large_data_reduce
  type: reduce
  reduce_key: category
  prompt: |
    Summarize the data for category {{ inputs[0].category }}:
    {% for item in inputs %}
    Item {{ loop.index }}: {{ item.data }}
    {% endfor %}
  fold_prompt: |
    Combine the following summaries for category {{ inputs[0].category }}:
    Current summary: {{ output.summary }}
```

```
      New data:
      {% for item in inputs %}
      Item {{ loop.index }}: {{ item.data }}
      {% endfor %}
    fold_batch_size: 100
    output:
      schema:
        summary: string
```

Example Rendered Prompt

Rendered Reduce Prompt

Let's consider an example of how a reduce operation prompt might look when rendered with actual data. Assume we have a reduce operation that summarizes product reviews, and we're processing reviews for a product with ID "PROD123". Here's what the rendered prompt might look like:

```
Summarize the reviews for product PROD123:

Review 1: This laptop is amazing! The battery life is incredible, lasting me a
full day of work without needing to charge. The display is crisp and vibrant,
perfect for both work and entertainment. The only minor drawback is that it can
get a bit warm during intensive tasks.

Review 2: I'm disappointed with this purchase. While the laptop looks sleek,
its performance is subpar. It lags when running multiple applications, and the
fan noise is quite noticeable. On the positive side, the keyboard is
comfortable to type on.

Review 3: Decent laptop for the price. It handles basic tasks well, but
struggles with more demanding software. The build quality is solid, and I
appreciate the variety of ports. Battery life is average, lasting about 6 hours
with regular use.

Review 4: Absolutely love this laptop! It's lightweight yet powerful, perfect
for my needs as a student. The touchpad is responsive, and the speakers produce
surprisingly good sound. My only wish is that it had a slightly larger screen.

Review 5: Mixed feelings about this product. The speed and performance are
great for everyday use and light gaming. However, the webcam quality is poor,
which is a letdown for video calls. The design is sleek, but the glossy finish
attracts fingerprints easily.
```

This example shows how the prompt template is filled with actual review data for a specific product. The language model would then process this prompt to generate a summary of the reviews for the product.

Scratchpad Technique

When doing an incremental reduce, the task may require intermediate state that is not represented in the output. For example, if the task is to determine all features more than one person liked about the product, we need some intermediate state to keep track of the features that have been liked once, so if we see the same feature liked again, we can update the output. DocETL maintains an internal "scratchpad" to handle this.

How it works

1. The process starts with an empty accumulator and an internal scratchpad.
2. It sequentially folds in batches of more than one element at a time.
3. The internal scratchpad tracks additional state necessary for accurately solving tasks incrementally. The LLM decides what to write to the scratchpad.
4. During each internal LLM call, the current scratchpad state is used along with the accumulated output and new inputs.
5. The LLM updates both the accumulated output and the internal scratchpad, which are used in the next fold operation.

The scratchpad technique is handled internally by DocETL, allowing users to define complex reduce operations without worrying about the complexities of state management across batches. Users provide their reduce and fold prompts focusing on the desired output, while DocETL uses the scratchpad technique behind the scenes to ensure accurate tracking of trends and efficient processing of large datasets.

Value Sampling

For very large groups, you can use value sampling to process a representative subset of the data. This can significantly reduce processing time and costs.

The following table outlines the available value sampling methods:

Method	Description
random	Randomly select a subset of values
first_n	Select the first N values
cluster	Use K-means clustering to select representative samples
semantic_similarity	Select samples based on semantic similarity to a query

To enable value sampling, add a `value_sampling` configuration to your reduce operation. The configuration should specify the method, sample size, and any additional parameters required by the chosen method.

Value Sampling Configuration

```yaml
- name: sampled_reduce
  type: reduce
  reduce_key: product_id
  prompt: |
    Summarize the reviews for product {{ inputs[0].product_id }}:
    {% for item in inputs %}
    Review {{ loop.index }}: {{ item.review }}
    {% endfor %}
  value_sampling:
    enabled: true
    method: cluster
    sample_size: 50
  output:
    schema:
      summary: string
```

In this example, the Reduce operation will use K-means clustering to select a representative sample of 50 reviews for each product_id.

For semantic similarity sampling, you can use a query to select the most relevant samples. This is particularly useful when you want to focus on specific aspects of the data.

Semantic Similarity Sampling

```yaml
- name: sampled_reduce_sem_sim
  type: reduce
  reduce_key: product_id
  prompt: |
    Summarize the reviews for product {{ inputs[0].product_id }}, focusing on
comments about battery life and performance:
    {% for item in inputs %}
    Review {{ loop.index }}: {{ item.review }}
    {% endfor %}
  value_sampling:
    enabled: true
    method: sem_sim
    sample_size: 30
    embedding_model: text-embedding-3-small
    embedding_keys:
      - review
    query_text: "Battery life and performance"
  output:
    schema:
      summary: string
```

In this example, the Reduce operation will use semantic similarity to select the 30 reviews most relevant to battery life and performance for each product_id. This allows you to focus the summarization on specific aspects of the product reviews.

Lineage

The Reduce operation supports lineage, which allows you to track the original input data for each output. This can be useful for debugging and auditing. To enable lineage, add a `lineage` configuration to your reduce operation, specifying the keys to include in the lineage. For example:

```
- name: summarize_reviews_by_category
  type: reduce
  reduce_key: category
  prompt: |
    Summarize the reviews for category {{ inputs[0].category }}:
    {% for item in inputs %}
    Review {{ loop.index }}: {{ item.review }}
    {% endfor %}
  output:
    schema:
      summary: string
    lineage:
      - product_id
```

This output will include a list of all product_ids for each category in the lineage, saved under the key `summarize_reviews_by_category_lineage`.

Best Practices

1. **Choose Appropriate Keys**: Select `reduce_key` (s) that logically group your data for the desired aggregation.
2. **Design Effective Prompts**: Create prompts that clearly instruct the model on how to aggregate or summarize the grouped data.
3. **Consider Data Size**: For large datasets, use incremental folding and value sampling to manage processing efficiently.
4. **Optimize Your Pipeline**: Use `docetl build pipeline.yaml` to optimize your pipeline, which can introduce efficient merge operations and resolve steps if needed.
5. **Balance Precision and Efficiency**: When dealing with very large groups, consider using value sampling to process a representative subset of the data.