Resolve Operation

The Resolve operation in DocETL identifies and canonicalizes duplicate entities in your data. It's particularly useful when dealing with inconsistencies that can arise from LLM-generated content, or data from multiple sources.

Motivation

Map operations executed by LLMs may sometimes yield inconsistent results, even when referring to the same entity. For example, when extracting patient names from medical transcripts, you might end up with variations like "Mrs. Smith" and "Jane Smith" for the same person. In such cases, a Resolve operation on the `patient_name` field can help standardize patient names before conducting further analysis.

🚀 Example: Standardizing Patient Names

Let's see a practical example of using the Resolve operation to standardize patient names extracted from medical transcripts.

```
- name: standardize_patient_names
  type: resolve
  optimize: true
  comparison_prompt: |
    Compare the following two patient name entries:

    Patient 1: {{ input1.patient_name }}
    Date of Birth 1: {{ input1.date_of_birth }}

    Patient 2: {{ input2.patient_name }}
    Date of Birth 2: {{ input2.date_of_birth }}

    Are these entries likely referring to the same patient? Consider name
similarity and date of birth. Respond with "True" if they are likely the same
patient, or "False" if they are likely different patients.
  resolution_prompt: |
    Standardize the following patient name entries into a single, consistent
format:

    {% for entry in inputs %}
    Patient Name {{ loop.index }}: {{ entry.patient_name }}
    {% endfor %}

    Provide a single, standardized patient name that represents all the
matched entries. Use the format "LastName, FirstName MiddleInitial" if
```

```
available.
  output:
    schema:
      patient_name: string
```

This Resolve operation processes patient names to identify and standardize duplicates:

1. Compares all pairs of patient names using the `comparison_prompt`. In the prompt, you can reference to the documents via `input1` and `input2`.
2. For identified duplicates, it applies the `resolution_prompt` to generate a standardized name. You can reference all matched entries via the `inputs` variable.

Note: The prompt templates use Jinja2 syntax, allowing you to reference input fields directly (e.g., `input1.patient_name`).

> ⚠ **Performance Consideration**
>
> You should not run this operation as-is unless your dataset is small! Running O(n^2) comparisons with an LLM can be extremely time-consuming for large datasets. Instead, optimize your pipeline first using `docetl build pipeline.yaml` and run the optimized version, which will generate efficient blocking rules for the operation. Make sure you've set `optimize: true` in your resolve operation config.

Blocking

To improve efficiency, the Resolve operation supports "blocking" - a technique to reduce the number of comparisons by only comparing entries that are likely to be matches. DocETL supports two types of blocking:

1. Embedding similarity: Compare embeddings of specified fields and only process pairs above a certain similarity threshold.
2. Python conditions: Apply custom Python expressions to determine if a pair should be compared.

Here's an example of a Resolve operation with blocking:

```
- name: standardize_patient_names
  type: resolve
  comparison_prompt: |
    # (Same as previous example)
  resolution_prompt: |
    # (Same as previous example)
  output:
    schema:
      patient_name: string
  blocking_keys:
```

```
    - last_name
    - date_of_birth
  blocking_threshold: 0.8
  blocking_conditions:
    - "left['last_name'][:2].lower() == right['last_name'][:2].lower()"
    - "left['first_name'][:2].lower() == right['first_name'][:2].lower()"
    - "left['date_of_birth'] == right['date_of_birth']"
    - "left['ssn'][-4:] == right['ssn'][-4:]"
```

In this example, pairs will be considered for comparison if:

- The embedding similarity of their `last_name` and `date_of_birth` fields is above 0.8, OR
- The `last_name` fields start with the same two characters, OR
- The `first_name` fields start with the same two characters, OR
- The `date_of_birth` fields match exactly, OR
- The last four digits of the `ssn` fields match.

How the Comparison Algorithm Works

After determining eligible pairs for comparison, the Resolve operation uses a Union-Find (Disjoint Set Union) algorithm to efficiently group similar items. Here's a breakdown of the process:

1. **Initialization**: Each item starts in its own cluster.
2. **Pair Generation**: All possible pairs of items are generated for comparison.
3. **Batch Processing**: Pairs are processed in batches (controlled by `compare_batch_size`).
4. **Comparison**: For each batch: a. An LLM performs pairwise comparisons to determine if items match. b. Matching pairs trigger a `merge_clusters` operation to combine their clusters.
5. **Iteration**: Steps 3-4 repeat until all pairs are compared.
6. **Result Collection**: All non-empty clusters are collected as the final result.

Efficiency

The batch processing of comparisons allows for efficient, incremental clustering as matches are found, without needing to rebuild the entire cluster structure after each match. This allows for parallelization of LLM calls, improving overall performance. However, this also limits parallelism to the batch size, so choose an appropriate value for `compare_batch_size` based on your dataset size and system capabilities.

Required Parameters

- `type`: Must be set to "resolve".
- `comparison_prompt`: The prompt template to use for comparing potential matches.
- `resolution_prompt`: The prompt template to use for reducing matched entries.
- `output`: Schema definition for the output from the LLM.

Optional Parameters

Parameter	Description	Default
`embedding_model`	The model to use for creating embeddings	Falls back to `default_model`
`resolution_model`	The language model to use for reducing matched entries	Falls back to `default_model`
`comparison_model`	The language model to use for comparing potential matches	Falls back to `default_model`
`blocking_keys`	List of keys to use for initial blocking	All keys in the input data
`blocking_threshold`	Embedding similarity threshold for considering entries as potential matches	None
`blocking_conditions`	List of conditions for initial blocking	[]
`input`	Specifies the schema or keys to subselect from each item to pass into the prompts	All keys from input items
`embedding_batch_size`	The number of entries to send to the embedding model at a time	1000
`compare_batch_size`	The number of entity pairs processed in each batch during the comparison phase	500

Parameter	Description	Default
`limit_comparisons`	Maximum number of comparisons to perform	None
`timeout`	Timeout for each LLM call in seconds	120
`max_retries_per_timeout`	Maximum number of retries per timeout	2
`sample`	Number of samples to use for the operation	None
`litellm_completion_kwargs`	Additional parameters to pass to LiteLLM completion calls.	{}
`bypass_cache`	If true, bypass the cache for this operation.	False

Best Practices

1. **Anticipate Resolve Needs**: If you anticipate needing a Resolve operation and want to control the prompts, create it in your pipeline and let the optimizer find the appropriate blocking rules and thresholds.
2. **Let the Optimizer Help**: The optimizer can detect if you need a Resolve operation (e.g., because there's a downstream reduce operation you're optimizing) and can create a Resolve operation with suitable prompts and blocking rules.
3. **Effective Comparison Prompts**: Design comparison prompts that consider all relevant factors for determining matches.
4. **Detailed Resolution Prompts**: Create resolution prompts that effectively standardize or combine information from matched records.
5. **Appropriate Model Selection**: Choose suitable models for embedding (if used) and language tasks.
6. **Optimize Batch Size**: If you expect to compare a large number of pairs, consider increasing the `compare_batch_size`. This parameter effectively limits parallelism, so a larger value can improve performance for large datasets.

Balancing Batch Size

While increasing `compare_batch_size` can improve parallelism, be cautious not to set it too high. Extremely large batch sizes might overwhelm system memory or exceed API rate limits. Consider your system's capabilities and the characteristics of your dataset when adjusting this parameter.

The Resolve operation is particularly useful for data cleaning, deduplication, and creating standardized records from multiple data sources. It can significantly improve data quality and consistency in your dataset.