Gather Operation

The Gather operation in DocETL is designed to maintain context when processing divided documents. It complements the Split operation by adding contextual information from surrounding chunks to each segment.

Motivation

When splitting long documents, such as complex legal contracts or court transcripts, individual chunks often lack sufficient context for accurate analysis or processing. This can lead to several challenges:

- Loss of reference information (e.g., terms defined in earlier sections)
- Incomplete representation of complex clauses that span multiple chunks
- Difficulty in understanding the broader document structure
- Missing important context from preambles or introductory sections

Context Challenge in Legal Documents

Imagine a lengthy merger agreement split into chunks. A single segment might contain clauses referencing "the Company" or "Effective Date" without clearly defining these terms. Without context from previous chunks, it becomes challenging to interpret the legal implications accurately.

How Gather Works

The Gather operation addresses these challenges by:

1. Identifying relevant surrounding chunks (peripheral context)
2. Adding this context to each chunk
3. Preserving document structure information

Peripheral Context

Peripheral context refers to the surrounding text or information that helps provide a more complete understanding of a specific chunk of content. In legal documents, this can

include:

- Preceding text that introduces key terms, parties, or conditions
- Following text that elaborates on clauses presented in the current chunk
- Document structure information, such as article or section headers
- Summarized versions of nearby chunks for efficient context provision

Document Structure

The Gather operation can maintain document structure through header hierarchies. This is particularly useful for preserving the overall structure of complex legal documents like contracts, agreements, or regulatory filings.

🚀 Example: Enhancing Context in Legal Document Analysis

Let's walk through an example of using the Gather operation to process a long merger agreement.

Step 1: Extract Metadata (Map operation before splitting)

First, we extract important metadata from the full document:

```yaml
- name: extract_metadata
  type: map
  prompt: |
    Extract the following metadata from the merger agreement:
    1. Agreement Date
    2. Parties involved
    3. Total value of the merger (if specified)

    Agreement text:
    {{ input.agreement_text }}

    Return the extracted information in a structured format.
  output:
    schema:
      agreement_date: string
      parties: list[string]
      merger_value: string
```

Step 2: Split Operation

Next, we split the document into manageable chunks:

```yaml
- name: split_merger_agreement
  type: split
  split_key: agreement_text
  method: token_count
  method_kwargs:
    token_count: 1000
```

Step 3: Extract Headers (Map operation)

We extract headers from each chunk:

```yaml
- name: extract_headers
  type: map
  input:
    - agreement_text_chunk
  prompt: |
    Extract any section headers from the following merger agreement chunk:
    {{ input.agreement_text_chunk }}
    Return the headers as a list, preserving their hierarchy.
  output:
    schema:
      headers: "list[{header: string, level: integer}]"
```

Step 4: Gather Operation

Now, we apply the Gather operation:

```yaml
- name: context_gatherer
  type: gather
  content_key: agreement_text_chunk
  doc_id_key: split_merger_agreement_id
  order_key: split_merger_agreement_chunk_num
  peripheral_chunks:
    previous:
      middle:
        content_key: agreement_text_chunk_summary
      tail:
        content_key: agreement_text_chunk
    next:
      head:
        count: 1
        content_key: agreement_text_chunk
  doc_header_key: headers
```

Step 5: Analyze Chunks (Map operation after Gather)

Finally, we analyze each chunk with its gathered context:

```yaml
- name: analyze_chunks
  type: map
```

```
  input:
    - agreement_text_chunk_rendered
    - agreement_date
    - parties
    - merger_value
  prompt: |
    Analyze the following chunk of a merger agreement, considering the
provided metadata:

    Agreement Date: {{ input.agreement_date }}
    Parties: {{ input.parties | join(', ') }}
    Merger Value: {{ input.merger_value }}

    Chunk content:
    {{ input.agreement_text_chunk_rendered }}

    Provide a summary of key points and any potential legal implications in
this chunk.
  output:
    schema:
      summary: string
      legal_implications: list[string]
```

This configuration:

1. Extracts important metadata from the full document before splitting
2. Splits the document into manageable chunks
3. Extracts headers from each chunk
4. Gathers context for each chunk, including:
5. Summaries of the chunks before the previous chunk
6. The full content of the previous chunk
7. The full content of the current chunk
8. The full content of the next chunk
9. Extracted headers for levels directly above headers in the current chunk, for structural context
10. Analyzes each chunk with its gathered context and the extracted metadata

Configuration

The Gather operation includes several key components:

- `type`: Always set to "gather"
- `doc_id_key`: Identifies chunks from the same original document
- `order_key`: Specifies the sequence of chunks within a group

- `content_key`: Indicates the field containing the chunk content
- `peripheral_chunks`: Specifies how to include context from surrounding chunks
- `doc_header_key` (optional): Denotes a field representing extracted headers for each chunk
- `sample` (optional): Number of samples to use for the operation

Peripheral Chunks Configuration

The `peripheral_chunks` configuration in the Gather operation is highly flexible, allowing users to precisely control how context is added to each chunk. This configuration determines which surrounding chunks are included and how they are presented.

Structure

The `peripheral_chunks` configuration is divided into two main sections:

1. `previous`: Defines how chunks preceding the current chunk are included.
2. `next`: Defines how chunks following the current chunk are included.

Each of these sections can contain up to three subsections:

- `head`: The first chunk(s) in the section.
- `middle`: Chunks between the `head` and `tail` sections.
- `tail`: The last chunk(s) in the section.

Configuration Options

For each subsection, you can specify:

- `count`: The number of chunks to include (for `head` and `tail` only).
- `content_key`: The key in the chunk data that contains the content to use.

Example Configuration

```yaml
peripheral_chunks:
  previous:
    head:
      count: 1
      content_key: full_content
    middle:
      content_key: summary_content
    tail:
      count: 2
      content_key: full_content
  next:
    head:
```

```
        count: 1
        content_key: full_content
```

This configuration would:

1. Include the full content of the very first chunk.
2. Include summaries of all chunks between the `head` and `tail` of the previous section.
3. Include the full content of 2 chunks immediately before the current chunk.
4. Include the full content of 1 chunk immediately after the current chunk.

Behavior Details

1. **Content Selection**: If a `content_key` is specified that's different from the main content key, it's treated as a summary. This is useful for including condensed versions of chunks in the `middle` section to save space. If no `content_key` is specified, it defaults to the main content key of the operation.
2. **Chunk Ordering**: For the `previous` section, chunks are processed in reverse order (from the current chunk towards the beginning of the document). For the `next` section, chunks are processed in forward order.
3. **Skipped Content**: If there are gaps between included chunks, the operation inserts a note indicating how many characters were skipped. Example: `[... 5000 characters skipped ...]`
4. **Chunk Labeling**: Each included chunk is labeled with its order number and whether it's a summary. Example: `[Chunk 5 (Summary)]` or `[Chunk 6]`

Best Practices

1. **Balance Context and Conciseness**: Use full content for immediate context (`head`) and summaries for `middle` sections to provide context without overwhelming the main content.
2. **Adapt to Document Structure**: Adjust the `count` for `head` and `tail` based on the typical length of your document sections.
3. **Use Asymmetric Configurations**: You might want more previous context than next context, or vice versa, depending on your specific use case.
4. **Consider Performance**: Including too much context can increase processing time and token usage. Use summaries and selective inclusion to optimize performance.

By leveraging this flexible configuration, you can tailor the Gather operation to provide the most relevant context for your specific document processing needs, balancing

completeness with efficiency.

Output

The Gather operation adds a new field to each input document, named by appending "_rendered" to the `content_key`. This field contains:

1. The reconstructed header hierarchy (if applicable)
2. Previous context (if any)
3. The main chunk, clearly marked
4. Next context (if any)
5. Indications of skipped content between contexts

Sample Output for Merger Agreement

```
agreement_text_chunk_rendered: |
  _Current Section:_ # Article 5: Representations and Warranties > ## 5.1
Representations and Warranties of the Company

  --- Previous Context ---
  [Chunk 17] ... summary of earlier sections on definitions and parties ...
  [... 500 characters skipped ...]
  [Chunk 18] The Company hereby represents and warrants to Parent and Merger
Sub as follows, except as set forth in the Company Disclosure Schedule:
  --- End Previous Context ---

  --- Begin Main Chunk ---
  5.1.1 Organization and Qualification. The Company is duly organized, validly
existing, and in good standing under the laws of its jurisdiction of
organization and has all requisite corporate power and authority to own, lease,
and operate its properties and to carry on its business as it is now being
conducted...
  --- End Main Chunk ---

  --- Next Context ---
  [Chunk 20] 5.1.2 Authority Relative to This Agreement. The Company has all
necessary corporate power and authority to execute and deliver this Agreement,
to perform its obligations hereunder, and to consummate the Merger...
  [... 750 characters skipped ...]
  --- End Next Context ---
```

Handling Document Structure

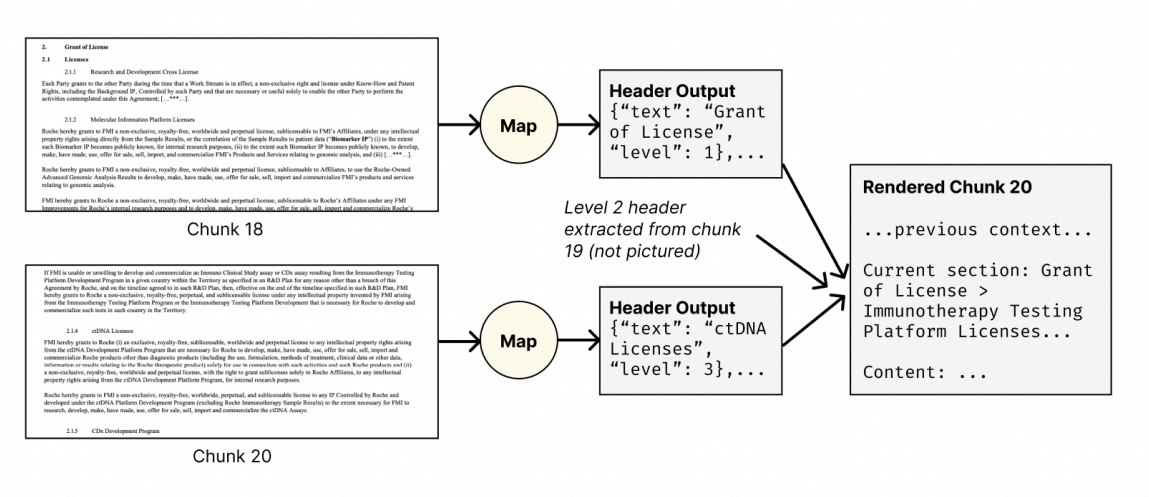
A key feature of the Gather operation is its ability to maintain document structure through header hierarchies. This is particularly useful for preserving the overall structure of complex documents like legal contracts, technical manuals, or research papers.

How Header Handling Works

1. Headers are typically extracted from each chunk using a Map operation after the Split but before the Gather.
2. The Gather operation uses these extracted headers to reconstruct the relevant header hierarchy for each chunk.
3. When rendering a chunk, the operation includes all the most recent headers from higher levels found in previous chunks.
4. This ensures that each rendered chunk includes a complete "path" of headers leading to its content, preserving the document's overall structure and context.

Example: Header Handling in Legal Contracts

Let's look at an example of how the Gather operation handles document headers in the context of legal contracts:



In this figure:

1. We see two chunks (18 and 20) from a 74-page legal contract.
2. Each chunk goes through a Map operation to extract headers.
3. For Chunk 18, a level 1 header "Grant of License" is extracted.
4. For Chunk 20, a level 3 header "ctDNA Licenses" is extracted.
5. When rendering Chunk 20, the Gather operation includes:
6. The level 1 header from Chunk 18 ("Grant of License")
7. A level 2 header from Chunk 19 (not pictured, but included in the rendered output)
8. The current level 3 header from Chunk 20 ("ctDNA Licenses")

This hierarchical structure provides crucial context for understanding the content of Chunk 20, even though the higher-level headers are not directly present in that chunk.

> **Importance of Header Hierarchy**
>
> By maintaining the header hierarchy, the Gather operation ensures that each chunk is placed in its proper context within the overall document structure. This is especially crucial for complex documents where understanding the relationship between different sections is key to accurate analysis or processing.

Best Practices

1. **Extract Metadata Before Splitting**: Run a map operation on the full document before splitting to extract any metadata that could be useful when processing any chunk (e.g., agreement dates, parties involved). Reference this metadata in subsequent map operations after the gather step.
2. **Balance Context and Efficiency**: For ultra-long documents, consider using summarized versions of chunks in the "middle" sections to strike a balance between providing context and managing the overall size of the processed data.
3. **Preserve Document Structure**: Utilize the `doc_header_key` parameter to include relevant structural information from the original document, which can be important for understanding the context of complex or structured content.
4. **Tailor Context to Your Task**: Adjust the `peripheral_chunks` configuration based on the specific needs of your analysis. Some tasks may require more preceding context, while others might benefit from more following context.
5. **Combine with Other Operations**: The Gather operation is most powerful when used in conjunction with Split, Map (for summarization or header extraction), and subsequent analysis operations to process long documents effectively.
6. **Consider Performance**: Be mindful of the increased token count when adding context. Adjust your downstream operations accordingly to handle the larger input sizes, and use summarized context where appropriate to manage token usage.
7. **Use `next` Sparingly**: The `next` parameter in the Gather operation should be used judiciously. It's primarily beneficial in specific scenarios:
   - When dealing with structured data or tables where the next chunk provides essential context for understanding the current chunk.
   - In cases where the end of a chunk might cut off a sentence or important piece of information that continues in the next chunk.

For most text-based documents, focusing on the `prev` context is usually sufficient. Overuse of `next` can lead to unnecessary token consumption and potential redundancy in the gathered output.

When to Use `next`

Consider using `next` when processing:

```
- Financial reports with multi-page tables
- Technical documents where diagrams span multiple pages
- Legal contracts where clauses might be split across chunk boundaries
```

By default, it's recommended to set `next=0` unless you have a specific need for forward context in your document processing pipeline.