perplexity

What is Chunking?

Chunking is the process of breaking down large datasets or pieces of information into smaller, manageable units called "chunks." These chunks are designed to simplify processing, enhance memory usage, and improve scalability across various applications, including artificial intelligence (AI), big data analytics, and knowledge management systems[1] [2] [3]. Chunking is widely used in structured data (e.g., databases) and unstructured data (e.g., text, images, videos), enabling efficient handling and analysis[2] [3].

Types of Chunking

Chunking can be implemented in several ways depending on the nature of the data and the use case:

- **Fixed-size chunking**: Divides data into equal-sized chunks, suitable for streaming and file storage systems[1].
- **Variable-size chunking**: Creates chunks of varying sizes based on patterns or content, ideal for deduplication tasks[1].
- **Logical chunking**: Breaks data into logical units like paragraphs or time intervals, preserving semantic meaning[1].
- **Dynamic chunking**: Adjusts chunk size dynamically based on system constraints like memory availability[1].
- **Content-based chunking**: Splits data according to specific patterns within the content for tasks like backup or retrieval[1].

Why Do We Chunk Data?

Chunking serves multiple purposes across different domains:

1. Optimizing Memory Usage

Chunking prevents memory overload by processing large datasets in smaller pieces. For example, machine learning models often train on batches of data rather than the entire dataset at once to avoid resource constraints[1] [4].

2. Improving Data Transfer

Breaking large files into chunks facilitates faster and more reliable data transmission. If an error occurs during transfer, only the affected chunk needs to be resent instead of the entire file[1] [4].

3. Enabling Parallel Processing

Chunking allows distributed systems to process chunks simultaneously across multiple nodes or processors, reducing overall processing time and improving efficiency. This is particularly beneficial in AI workflows like training large models or indexing datasets for retrieval-augmented generation (RAG)[1] [4].

4. Enhancing Retrieval Accuracy

In applications like semantic search and RAG frameworks, chunking ensures that only relevant pieces of a document are indexed and retrieved for queries. Smaller chunks maintain focused context, improving precision and reducing noise in results[5] [6].

5. Scaling Systems

Chunking supports scalability by enabling systems to handle growing datasets more efficiently. It also allows for adaptive strategies that adjust to changing workloads or constraints[3] [4].

6. Preserving Semantic Integrity

For tasks involving natural language processing (NLP), maintaining semantic coherence within chunks is crucial. Techniques like sentence-based or smart chunking ensure that chunks retain meaningful context without fragmenting ideas[7] [6].

Practical Applications

Chunking is widely applied in:

- **Machine learning**: Batch processing during model training.
- **Vector databases**: Storing embeddings of smaller text segments for efficient retrieval.
- **RAG frameworks**: Dividing documents into semantically coherent chunks for precise query responses.
- **Streaming platforms**: Breaking videos into segments for adaptive streaming.
- **Knowledge management systems**: Organizing unstructured data into usable formats.

Considerations for Effective Chunking

When implementing chunking strategies, consider:

- **Chunk size**: Balancing between too small (causing inefficiencies) and too large (losing relevance)[1] [4].
- **Data type**: Structured vs. unstructured data may require different approaches[2] [3].
- **Processing environment**: Hardware limitations like memory capacity influence chunk size selection[1].
- **Semantic coherence**: Ensuring that chunks retain meaningful context for NLP tasks or search applications[6].

By leveraging appropriate chunking techniques, organizations can optimize their workflows, improve system performance, and enhance the usability of their datasets across diverse applications.

⁂

What factors influence chunk size?

Several factors influence the optimal chunk size in data processing, retrieval-augmented generation (RAG) systems, and other AI applications. These factors ensure efficient resource utilization, accurate retrieval, and effective system performance. Below are the key considerations:

Factors Influencing Chunk Size

1. Memory Constraints

- Larger chunks require more memory during processing, which can lead to system overload if memory resources are limited.
- Smaller chunks help prevent memory bottlenecks and allow for parallel processing but may increase overhead due to frequent task switching[8] [9].

2. Nature of the Content

- The type of data being processed plays a significant role. For example:
  - Long documents (e.g., books or articles) may benefit from larger chunks to retain context.
  - Shorter content (e.g., tweets or instant messages) requires smaller chunks for semantic granularity[10] [11] [12].

3. Embedding Model Constraints

- Different embedding models have token limits that dictate the maximum chunk size. For instance:
  - Sentence-transformer models perform well on individual sentences.
  - Models like `text-embedding-ada-002` work optimally with chunks containing 256–512 tokens[10] [13].

4. Query Complexity

- The expected complexity of user queries influences chunk size:
  - Short and specific queries benefit from smaller chunks to avoid noise.
  - Complex queries requiring broader context may need larger chunks to capture sufficient information[11] [14].

5. Granularity vs Context

- Smaller chunks improve retrieval precision by focusing on specific details but may lose broader context.
- Larger chunks provide comprehensive context but risk diluting relevance by including extraneous information[15] [16] [13].

6. Expected Access Patterns

- The structure of the data and how it will be accessed influence chunk size:
  - For time-series data, chunking might prioritize temporal dimensions over spatial ones.
  - Applications requiring both spatial and temporal access may need balanced chunk shapes[8] [12].

7. Computational Efficiency

- Larger chunks reduce the number of retrieval queries but increase computational load during processing.
- Smaller chunks require frequent queries, which can lead to higher computational overhead but faster response times in real-time applications[16] [9].

8. Task-Specific Requirements

- Different tasks require varying chunk sizes:
  - Semantic search benefits from smaller chunks for precise matching.
  - Summarization tasks may favor larger chunks for capturing broader context[10] [11].

9. Text Structure

- The structure of the text (e.g., sentences, paragraphs, tables) impacts chunking strategies:
  - Sentence-based chunking is ideal for sentence-transformer models.
  - Paragraph-based chunking works well for tasks requiring contextual coherence[12].

10. Compression and Storage Considerations

- Larger chunks reduce storage overhead by minimizing the number of entries in databases but may lead to inefficiencies in retrieval accuracy.
- Compression schemes like LZ4 or Zstd+Bitshuffle can influence optimal chunk size by balancing speed and storage efficiency[8] [17].

Balancing Trade-offs

Determining the ideal chunk size involves balancing trade-offs between precision, context retention, computational efficiency, and resource constraints. Experimentation with various chunk sizes and evaluation metrics (e.g., precision, recall, relevancy) is often necessary to identify the best configuration for a specific use case[15] [13] [16].

⁂

Types of Chunking Methods

Different chunking methods are designed to address specific use cases and data characteristics. Below is an overview of commonly used chunking strategies:

1. Fixed-Size Chunking

- **Description**: Divides text into uniform chunks based on a predefined size, such as token count, word count, or character count.
- **Advantages**:
  - Simple and computationally efficient.
  - Predictable chunk sizes, making it suitable for storage and indexing.
- **Disadvantages**:
  - May split sentences or paragraphs, disrupting semantic meaning.
  - Rigid and doesn't adapt to the natural flow of text.
- **Use Cases**: Ideal for tasks requiring consistent input sizes, such as feeding data into machine learning models with fixed input dimensions[18] [19] [20].

2. Sentence-Based Chunking

- **Description**: Segments text based on sentence boundaries using NLP libraries (e.g., spaCy or NLTK).
- **Advantages**:
  - Preserves grammatical and contextual integrity.
  - Aligns with human-readable structures for better interpretability.
- **Disadvantages**:
  - May result in variable chunk sizes, which could complicate processing.
- **Use Cases**: Suitable for tasks like translation, sentiment analysis, or summarization where sentence coherence is critical[19] [21].

3. Recursive Chunking

- **Description**: Iteratively splits text using hierarchical separators (e.g., paragraphs, sentences, words) until chunks meet the desired size or structure.
- **Advantages**:
  - Flexible and ensures size compliance.
  - Handles variable-length text gracefully.
- **Disadvantages**:
  - Over-splitting can disrupt coherence.
  - Multiple recursion levels increase processing time.
- **Use Cases**: Useful for structured documents like Python code or academic papers where logical boundaries must be preserved[18] [19] [22].

4. Semantic Chunking

- **Description**: Uses machine learning and NLP techniques to create chunks based on semantic coherence, topic continuity, or linguistic cues.
- **Advantages**:
  - Produces semantically rich and contextually relevant chunks.
  - Reduces noise by focusing on meaningful content.
- **Disadvantages**:
  - Computationally expensive due to reliance on advanced NLP models.
- **Use Cases**: Ideal for retrieval-augmented generation (RAG) systems where semantic relevance is crucial[23] [24].

5. Sliding Window Chunking

- **Description**: Creates overlapping chunks by sliding a fixed-size window across the text. A portion of the previous chunk is repeated in the next one to preserve context.
- **Advantages**:
  - Maintains context across chunk boundaries.
  - Reduces loss of meaning at edges of chunks.
- **Disadvantages**:
  - Increases redundancy and storage requirements.
- **Use Cases**: Effective for tasks requiring contextual continuity, such as conversational AI or document chat systems[21] [20].

6. Specialized Chunking

- **Description**: Tailored methods for structured formats like Markdown or LaTeX that preserve the original layout and hierarchy during chunking.
- **Advantages**:
  - Ensures logical organization of content (e.g., sections, headings).
  - Improves accuracy for technical documents or codebases.
- **Disadvantages**:
  - Requires domain-specific parsing tools (e.g., MarkdownTextSplitter).
- **Use Cases**: Suitable for technical documents, academic papers, or structured content[18] [22].

7. Token-Based Chunking

- **Description**: Splits text based on token counts rather than characters or words. Often used with models that have token limits (e.g., GPT models).
- **Advantages**:
  - Adheres to model constraints like maximum token limits.
  - Ensures compatibility with embedding-based applications.
- **Disadvantages**:
  - May disrupt semantic meaning if tokens are split mid-sentence.
- **Use Cases**: Common in LLM applications where token limits dictate input size[24] [25].

8. Smart Chunking

- **Description**: Dynamically determines segmentation based on semantic similarity, topic continuity, and linguistic cues using clustering techniques.
- **Advantages**:
  - Produces highly meaningful and coherent chunks tailored to underlying themes or patterns in the data.
  - Reduces manual preprocessing effort by automating segmentation decisions.
- **Disadvantages**:
  - Computationally intensive due to reliance on advanced machine learning models.
- **Use Cases**: Useful for customer feedback analysis, market research, and trend analysis[23] [26].

Additional Methods

Other approaches include hybrid strategies (combining multiple methods), distributed chunking (splitting across nodes for scalability), and logical chunking (preserving scene boundaries in videos)[27].

By selecting the appropriate chunking method based on data type, task requirements, and resource constraints, organizations can optimize their workflows and improve performance in applications like search systems, RAG frameworks, and LLM-based solutions.

⁂

How to find the ideal chunk size?

Finding the ideal chunk size for a system, particularly in retrieval-augmented generation (RAG) or similar applications, involves balancing several factors to optimize performance, accuracy, and efficiency. Below is a step-by-step guide to determining the optimal chunk size based on the provided search results:

Steps to Find the Ideal Chunk Size

1. Preprocess Your Data

Before determining chunk size, ensure your data is clean and free from noise. For example:

- Remove unnecessary elements like HTML tags or irrelevant metadata.
- Ensure the text is well-structured for logical splitting (e.g., sentences, paragraphs)[28] [29].

2. Understand Your Use Case

The optimal chunk size depends heavily on the specific application:

- **Semantic Search**: Smaller chunks improve precision by focusing on granular details.
- **Question Answering**: Larger chunks may help retain more context for complex queries.
- **Summarization**: Larger chunks are better for capturing broader context[28] [30] [31].

3. Consider the Nature of the Content

The type of content influences chunk size:

- **Short Texts**: For tweets or code snippets, smaller chunks (e.g., 128–256 tokens) are appropriate.
- **Long Documents**: For articles or books, larger chunks (e.g., 512–1024 tokens) may be necessary to preserve context[28] [32].

4. Account for Model Constraints

Different embedding models have token limits and perform optimally at specific chunk sizes:

- Sentence-transformer models work well with sentence-level chunks.
- Models like `text-embedding-ada-002` perform better with 256–512 tokens per chunk[28] [32].
- Ensure that chunk sizes align with the model's maximum context window if results will be fed into another LLM[28] [31].

5. Experiment with Overlap

Introduce overlapping tokens between chunks to preserve context across boundaries:

- A typical overlap is 10–20% of the chunk size (e.g., 100 tokens overlap for a 512-token chunk).
- Overlapping ensures continuity but increases storage and computational requirements[30] [31].

6. Test Across a Range of Chunk Sizes

Evaluate different sizes to find the balance between granularity and context retention:

- Start with smaller sizes (e.g., 128, 256 tokens) and scale up incrementally (e.g., 512, 1024 tokens).
- Use tools like LangChain or LlamaIndex to automate testing across multiple chunk sizes[28] [33].

7. Evaluate Performance Using Metrics

Test each chunk size using a representative dataset and evaluate using metrics such as:

- **Relevance**: How well do retrieved chunks match user queries?
- **Faithfulness**: Does the system avoid hallucinations and stay grounded in source data?
- **Response Time**: How quickly does the system generate responses? Larger chunks may slow down processing[34] [33].

For example:

```
chunk_sizes = [128, 256, 512, 1024]
for size in chunk_sizes:
    avg_response_time, avg_faithfulness, avg_relevancy = evaluate_response_time_and_accu
    print(f"Chunk Size {size}: Response Time={avg_response_time}s, Faithfulness={avg_fait
```

8. Balance Trade-offs

Finding the ideal chunk size involves balancing competing priorities:

- Smaller chunks improve precision but may lose broader context.
- Larger chunks retain more context but risk diluting relevance and slowing down response times[30] [34].

9. Iterate and Refine

This process is iterative—test different configurations until you achieve optimal results for your use case. Use feedback from real-world performance to fine-tune further[28] [35].

Key Takeaways

To determine the ideal chunk size:

1. Preprocess your data for quality.
2. Consider your use case and content type.
3. Align with model constraints and token limits.
4. Experiment with overlaps to preserve context.
5. Test various sizes using metrics like relevance, faithfulness, and response time.

By following these steps, you can identify a chunking strategy that balances precision, efficiency, and contextual richness tailored to your application's needs.

⁂

What is the best method to digitize and chunk complex documents like annual reports?

To digitize and chunk complex documents like annual reports effectively, you need a combination of robust digitization techniques and advanced chunking strategies tailored to the structure and content of the document. Here's a detailed approach:

1. Digitization Process

Before chunking, annual reports must be converted into machine-readable formats. The following steps are essential:

Step 1: Document Scanning

- Use high-quality scanners to create digital copies of physical documents.
- Ensure scans are clear and free of distortions to maintain data integrity[36].

Step 2: Optical Character Recognition (OCR)

- Apply OCR tools (e.g., Tesseract, ABBYY FineReader) to extract text from scanned PDFs or images.
- For complex layouts (e.g., tables, charts), use Intelligent Document Processing (IDP) tools that can handle nested elements like tables and images[36] [37].

Step 3: Quality Assurance

- Validate the accuracy of the digitized text by comparing it with the original document.
- Check for errors such as missing data, misinterpreted characters, or misaligned tables[36].

2. Chunking Strategies for Annual Reports

Annual reports are highly structured documents with sections like executive summaries, financial statements, and governance details. The ideal chunking method should preserve this structure while optimizing for retrieval and analysis.

A. Structural Element-Based Chunking

- **Description**: Divide the document into chunks based on structural elements such as titles, sections, tables, and figures.
- **Method**:
  - Identify key sections like "Executive Summary," "Financial Statements," or "Risk Factors" using document understanding models.
  - Start a new chunk at each title or table element while preserving their integrity.
  - Merge smaller elements (e.g., short paragraphs) until a desired chunk size is reached[38].
- **Advantages**:
  - Retains logical organization and context.
  - Ensures important sections remain intact for accurate retrieval.

B. Semantic Chunking

- **Description**: Split text based on semantic coherence by analyzing sentence embeddings.
- **Method**:
  - Use NLP models to calculate semantic similarity between consecutive sentences.

    - Split chunks when there is a significant change in meaning (e.g., cosine similarity threshold)[39] [40].
- **Advantages**:
    - Produces coherent chunks that align with topics or themes.
    - Improves retrieval precision by avoiding fragmented context.

C. Token-Based Chunking

- **Description**: Divide text into chunks based on a fixed token limit (e.g., 512 tokens).
- **Method**:
    - Count tokens using tokenizers compatible with your language model (e.g., GPT tokenizers).
    - Ensure chunks do not exceed model constraints[39] [41].
- **Advantages**:
    - Guarantees compatibility with LLMs that have token limits.
    - Simplifies implementation.

D. Recursive Chunking

- **Description**: Iteratively split text at larger boundaries (e.g., sections) and refine into smaller chunks if needed.
- **Method**:
    - Start with larger splits (e.g., paragraphs).
    - If a paragraph exceeds the size limit, split it further into sentences or tokens[41] [40].
- **Advantages**:
    - Adapts flexibly to varying document structures.
    - Preserves natural language boundaries.

E. Metadata-Enriched Chunking

- **Description**: Add metadata (e.g., section titles, summaries) to each chunk for improved indexing and retrieval.
- **Method**:
    - Use LLMs to generate metadata such as keywords or summaries for each chunk.
    - Prepend this metadata to the chunk before indexing[40].
- **Advantages**:
    - Enhances retrieval accuracy by providing additional context.
    - Useful for complex financial filings where disambiguation is critical.

3. Best Practices for Chunking Annual Reports

1. **Preserve Contextual Integrity**:
   - Avoid splitting tables or charts across chunks as they lose meaning when fragmented[38].
   - Use overlapping tokens between chunks to maintain continuity.
2. **Leverage Document Structure**:
   - Utilize headers, footers, and section titles as natural breakpoints[40].
   - For Markdown-based reports, split by headers like "MD&A" or "Risk Factors"[40].
3. **Optimize Chunk Size**:
   - Balance between too small (losing context) and too large (exceeding model constraints).
   - Typical sizes range from 256–512 tokens for LLM applications[39] [41].
4. **Test Retrieval Performance**:
   - Evaluate chunking strategies using metrics like relevance, precision, and recall in retrieval tasks.
5. **Automate with Tools**:
   - Use frameworks like LangChain or libraries like Unstructured.io to automate chunking processes[42] [38].

Example Workflow

1. Digitize the report using OCR/IDP tools.
2. Parse the document into structural elements (titles, paragraphs, tables).
3. Choose a chunking strategy (e.g., structural element-based or semantic).
4. Generate metadata for each chunk if needed.
5. Index the chunks in a vector database for retrieval.

By combining robust digitization techniques with advanced chunking strategies tailored to annual reports' structure, you can ensure efficient storage, accurate retrieval, and meaningful analysis of these complex documents.

⁂

How to handle tables during chunking?

Handling tables during chunking requires preserving their structure and context while ensuring they fit within the constraints of the chunking strategy. Below are effective methods and considerations for chunking tables based on the provided information:

Challenges in Chunking Tables

1. **Loss of Context**: Splitting a table without including headers can make rows meaningless.
2. **Row/Column Splits**: Breaking rows or columns across chunks disrupts data integrity.
3. **Invalid Formats**: If tables are represented in formats like JSON or XML, splitting them incorrectly can result in invalid data structures.
4. **Redundant Headers**: Repeating headers excessively across chunks can negatively impact search results and increase storage overhead.

Best Practices for Chunking Tables

1. Preserve Table Headers

- Always include table headers in each chunk to provide context for the rows.
- For large tables that span multiple chunks, repeat the headers in subsequent chunks to ensure interpretability[43] [44].

2. Chunk by Rows

- If the entire table cannot fit into a single chunk:
  - Split the table row by row, ensuring each chunk contains as many rows as possible without exceeding size limits.
  - Avoid splitting rows mid-record to maintain data integrity[43] [44].
- Example:
  - If a table has 100 rows and the chunk size allows 20 rows, split it into 5 chunks with 20 rows each.

3. Handle Wide Tables (Many Columns)

- For tables with many columns that exceed the chunk size:
  - Relax the chunk size temporarily to accommodate a single row with all its columns.
  - If even a single row exceeds the maximum size for embeddings, split it into smaller chunks by grouping columns logically[43].

4. Use Markdown or Structured Formats

- Convert tables into markdown or other structured formats (e.g., JSON) before chunking to preserve readability and structure.
- Markdown formatting ensures compatibility with retrieval systems and maintains semantic coherence[43].

5. Leverage Layout-Aware Parsing

- Use layout-aware parsers (e.g., Agentspace Enterprise) to detect and isolate table elements during preprocessing.
- These parsers identify tables as distinct entities and ensure they are treated separately from other document elements[45].

6. Apply Overlap for Context

- Introduce overlapping rows between chunks to maintain continuity across splits.
- This is particularly useful when tables contain related data spread across multiple rows[44] [46].

Recommended Strategies for Table Chunking

Method	Description	Advantages	Use Cases
Row-Based Chunking	Splits tables by rows, keeping headers intact in each chunk.	Preserves data integrity; avoids splitting records mid-row.	Small-to-medium-sized tables with many rows.
Column-Based Splitting	Splits wide tables by grouping columns logically when a single row is too large.	Handles wide tables effectively; retains semantic meaning of grouped data.	Tables with many columns exceeding embedding limits.
Markdown Formatting	Converts tables into markdown format for easier rendering and chunking.	Ensures compatibility with retrieval systems; improves readability.	Tables intended for use in retrieval-augmented generation (RAG).
Layout Parsing	Uses tools like layout-aware parsers to detect and isolate table elements.	Automates table detection; ensures structural coherence during chunking.	Complex documents with mixed content (e.g., text, images, tables).

Tools and Frameworks

1. **LangChain or** Unstructured.io: Provides APIs for handling table-specific chunking strategies like `basic` or `by_title`[44] [47].
2. **Agentspace Enterprise**: Offers layout-aware document parsing to detect and preserve table structures during chunking[45].
3. **Custom Table Chunkers**: Tools like Ragie's specialized table chunker ensure that headers are preserved, rows are not split mid-record, and large tables are handled efficiently[43].

By following these strategies, you can effectively handle tabular data during chunking while preserving its structure, integrity, and relevance for downstream tasks like search or retrieval-augmented generation (RAG).

⁂

How can I ensure table headers are not lost during chunking

To ensure table headers are not lost during chunking, you can adopt specialized strategies that preserve the structure and context of tables. Here are the methods based on best practices and insights from the search results:

1. Include Headers in Each Chunk

- Always repeat the table headers in every chunk that contains rows from the table. This ensures that each chunk retains contextual information about the data it contains[48] [49] [50].
- For example:
  - If a table is split into multiple chunks row-by-row, prepend the header row to each chunk.

2. Chunk by Rows

- Process tables row-by-row to create chunks, ensuring that no individual row is split across chunks. This maintains the integrity of each record while including the header row in every chunk[48] [49].
- Example:
  - If a table has 100 rows and the chunk size allows for 20 rows, create 5 chunks:
    - Chunk 1: Header + Rows 1–20
    - Chunk 2: Header + Rows 21–40, and so on.

3. Relax Chunk Size for Wide Tables

- For tables with many columns, if a single row exceeds the maximum chunk size:
  - Temporarily relax the chunk size to accommodate the full row along with its headers.
  - If this is not possible (e.g., due to model constraints), split large rows into smaller logical groups of columns[48].

4. Use Markdown Formatting

- Convert tables into markdown format during preprocessing. Markdown preserves headers and rows in a structured format, making it easier to handle tables during chunking without losing context[48] [49].
- Example:

```
| Header 1 | Header 2 | Header 3 |
|----------|----------|----------|
```

```
| Row 1    | Data     | Data     |
| Row 2    | Data     | Data     |
```

5. Introduce Overlap for Context

- Use overlapping rows between chunks to maintain continuity across splits. For example:
  - Add a few rows from the previous chunk to the next one while ensuring that headers are repeated[51] [50].

6. Use Layout-Aware Parsers

- Tools like Ragie's specialized table chunker or Unstructured.io's `by_title` strategy can automatically detect and preserve headers when splitting tables into chunks[48] [49].
- These tools ensure:
  - Headers are included in every chunk.
  - Rows remain intact and are not split mid-record.
  - Excessive repetition of headers is avoided.

7. Preprocess Tables as Separate Entities

- Treat tables as distinct elements during preprocessing and handle them separately from other document content.
- Extract tables into structured formats (e.g., JSON, CSV) before applying chunking strategies. This helps maintain their integrity and ensures headers are preserved[48] [50].

8. Evaluate Chunking Results

- After applying a chunking strategy, validate that:
  - Every chunk containing table data includes the header row.
  - No rows or columns are split across chunks.
- Use tools like LangChain or RAG evaluation frameworks to test retrieval accuracy and ensure no critical context is lost.

By following these methods, you can ensure that table headers remain intact during chunking, preserving both context and data integrity for downstream tasks like retrieval or semantic search.

⁂

What strategies can prevent table rows from being split across chunks

Preventing table rows from being split across chunks is essential for preserving the integrity and context of tabular data during processing. Below are strategies to achieve this effectively:

Strategies to Prevent Table Row Splitting

1. Include Entire Rows in Each Chunk

- Ensure that each chunk contains complete rows, including all columns. Avoid splitting rows mid-record, as this disrupts the logical structure of the table.
- **Implementation**:
  - During chunking, check the size of the row (e.g., token count or memory usage) and include it entirely in the current chunk if it fits within the constraints.

2. Repeat Headers Across Chunks

- Include table headers in every chunk that contains rows from the table. This ensures that rows retain their context even when split across multiple chunks.
- **Example**:
  - Chunk 1: Header + Rows 1–10
  - Chunk 2: Header + Rows 11–20

3. Adjust Chunk Size Dynamically

- Dynamically adjust chunk size to accommodate large rows or tables. If a row exceeds the predefined chunk size, temporarily increase the limit to include the entire row.
- **Implementation**:
  - Use a conditional check during chunk creation to ensure no row is partially included.

4. Use Logical Grouping

- Group related rows together based on logical identifiers (e.g., "ID" or "Category") before splitting into chunks. This ensures that all related rows stay within the same chunk.
- **Example**:
  - For a dataset with an "ID" column, group rows by ID and chunk them together.

5. Leverage Overlapping Rows

- Introduce overlapping rows between chunks to maintain continuity across splits while ensuring no row is fragmented.
- **Implementation**:
  - Add a few rows from the previous chunk to the next one, ensuring headers are repeated.

6. Use Layout-Aware Parsers

- Employ layout-aware tools like LangChain or Unstructured.io that can detect and process tables as distinct entities, ensuring rows are not split across chunks.
- These tools can automatically preserve headers and keep rows intact during chunking.

7. Wrap Tables in Containers

- Wrap tables inside logical containers (e.g., nested tables or blocks) that prevent splitting during processing.
- **Implementation (PDF)**:
  - In libraries like iText, use `PdfPTable.setKeepTogether(true)` to keep entire tables on one page or in one chunk.

8. Preprocess Tables Separately

- Extract tables from documents and preprocess them as distinct entities before applying chunking strategies. This allows for specialized handling of tabular data.
- Convert tables into structured formats like JSON or CSV for easier manipulation.

Tools and Frameworks

1. **LangChain**: Supports table-aware chunking strategies to ensure headers and rows are preserved.
2. **iText Library**: Offers features like `setKeepTogether(true)` and nested table wrapping to prevent row splitting in PDFs.
3. **Custom Parsers**: Build custom logic for detecting and processing tables separately from other document elements.

By implementing these strategies, you can ensure that table rows remain intact during chunking, preserving their structure and contextual relevance for downstream applications like retrieval or semantic search.

⁂

How do you handle very large table for better retrieval?

Handling very large tables for better retrieval requires strategies that optimize storage, indexing, and query execution while maintaining scalability and performance. Here are effective methods:

1. Partitioning

Partitioning divides large tables into smaller, manageable segments based on specific criteria (e.g., date ranges, geographic locations, or categorical values). This reduces the amount of data scanned during queries and improves retrieval speed.

Types of Partitioning:

- **Range Partitioning**: Splits rows based on value ranges (e.g., dates).
- **List Partitioning**: Groups rows by predefined lists of column values.
- **Hash Partitioning**: Uses a hash function to distribute rows across partitions.

Advantages:

- Enables partition elimination, where only relevant partitions are scanned during queries.
- Improves query parallelism by distributing processing across multiple partitions[52] [53].

2. Indexing

Indexes create pointers to specific rows or columns in a table, enabling faster data retrieval. Properly indexing frequently queried columns is critical for performance.

Types of Indexes:

- **Clustered Index**: Organizes rows on disk based on the index order; ideal for columns frequently queried.
- **Non-clustered Index**: Creates a separate structure for faster lookups.
- **Full-text Index**: Optimized for text-based searches in large datasets.

Best Practices:

- Index columns used in `WHERE`, `JOIN`, and `ORDER BY` clauses.
- Use composite indexes for multi-column queries[54] [53].

3. Horizontal Sharding

Horizontal sharding distributes rows of a table across multiple servers based on a sharding key (e.g., user ID or geographic region). This reduces the load on individual servers and enables parallel processing.

Advantages:

- Infinite scalability by adding more shards as data grows.
- Reduces contention and improves query speed[55].

4. Archiving Old Data

Move historical or rarely accessed data to separate storage systems to reduce the size of active tables. This minimizes query latency for current data.

Techniques:

- Use a "sliding window" approach to maintain a rolling time window for active data while archiving older partitions.
- Archive old partitions to cold storage or secondary databases[52] [53].

5. Query Optimization

Optimize SQL queries to limit data retrieval and improve performance:

- Use `LIMIT` and `OFFSET` clauses to paginate results.
- Replace nested subqueries with Common Table Expressions (CTEs) or optimized joins.
- Filter data early using `WHERE` conditions[54] [56].

6. Materialized Views

Materialized views store precomputed query results, which can be retrieved quickly without scanning the original table repeatedly.

Use Cases:

- Queries involving complex aggregations or calculations.
- Frequently executed queries where real-time updates are not required[57].

7. Big Data Frameworks

Leverage big data platforms like Apache Hadoop, Apache Spark, or Google BigQuery for distributed processing and efficient handling of massive datasets.

Features:

- Distributed storage and fault tolerance (e.g., Hadoop Distributed File System).
- In-memory processing for faster analytics (e.g., Apache Spark)[58] [59].

8. Monitoring and Maintenance

Regularly monitor database performance and update statistics to ensure efficient query execution plans. Perform maintenance tasks such as rebuilding indexes or reorganizing partitions[53].

Example Workflow for Handling Large Tables:

1. Partition the table based on a logical key (e.g., date).
2. Create clustered and non-clustered indexes on frequently queried columns.
3. Archive old data to separate storage systems periodically.
4. Optimize SQL queries using pagination, filtering conditions, and materialized views.
5. Scale horizontally using sharding as data volume grows.

By combining these strategies, you can handle very large tables efficiently while improving retrieval speed and maintaining scalability across different workloads.

⁂

How to handle list item during chunking?

To handle list items effectively during chunking, the goal is to preserve their semantic structure and context while ensuring they fit within size constraints. Below are strategies and best practices for chunking list items:

Strategies for Chunking List Items

1. Consolidate Entire Lists into a Single Chunk

- **Description**: Combine all list items into one chunk alongside any leading paragraph or introductory text that provides context for the list.
- **Advantages**:
  - Maintains semantic coherence by keeping related items together.

    - Prevents loss of context caused by splitting list items across chunks.
- **Implementation**:
    - Use a chunking function that identifies lists and groups them as a single unit.
    - Include any preceding text that introduces the list.

2. Preserve Hierarchical Structure

- **Description**: For nested lists (lists with sublists), ensure that parent-child relationships are preserved within the same chunk.
- **Advantages**:
    - Retains the logical flow of information.
    - Prevents confusion caused by separating sublists from their parent items.
- **Implementation**:
    - Use recursive chunking strategies to group nested elements hierarchically.
    - Split only when the entire hierarchical structure exceeds size limits.

3. Use Overlap for Context Continuity

- **Description**: Introduce overlapping tokens between chunks when splitting large lists to preserve continuity across chunks.
- **Advantages**:
    - Reduces semantic loss at chunk boundaries.
    - Improves retrieval accuracy by maintaining context across splits.
- **Implementation**:
    - Set an overlap parameter (e.g., `chunk_overlap=20`) to repeat part of the previous chunk in the next one.

4. Combine Small Lists

- **Description**: Merge small lists with surrounding text or other small sections to create chunks of optimal size without exceeding constraints.
- **Advantages**:
    - Prevents creation of excessively small chunks, which may dilute retrieval relevance.
    - Maximizes efficiency in storage and indexing.
- **Implementation**:
    - Use parameters like `combine_text_under_n_chars` to merge small lists with adjacent content.

5. Respect Structural Boundaries

- **Description**: Ensure that lists are not split mid-item or mid-sentence, even if they exceed size constraints. Instead, isolate oversized lists and handle them separately.
- **Advantages**:
  - Preserves the readability and integrity of list items.
  - Avoids creating incoherent chunks that disrupt comprehension.
- **Implementation**:
  - Use text-splitting techniques to isolate oversized lists into dedicated chunks.

6. Specialized Chunking for Markdown or Structured Formats

- **Description**: For documents in Markdown or other structured formats, use specialized chunkers that recognize list syntax (e.g., bullet points or numbered items).
- **Advantages**:
  - Maintains formatting and structure during chunking.
  - Improves rendering and retrieval accuracy for structured content.
- **Implementation**:

```
from langchain.text_splitter import MarkdownTextSplitter
markdown_text = "..."
markdown_splitter = MarkdownTextSplitter(chunk_size=100, chunk_overlap=0)
docs = markdown_splitter.create_documents([markdown_text])
```

Best Practices

1. **Preserve Contextual Integrity**:
   - Always include introductory paragraphs or lead-in sentences with the list items to maintain context.
2. **Avoid Fragmentation**:
   - Do not split individual list items across chunks, as this disrupts their meaning.
3. **Optimize Chunk Size**:
   - Balance between keeping lists intact and adhering to token or character limits (e.g., `max_characters=500`).
4. **Use Metadata for Retrieval**:
   - Add metadata (e.g., "List Heading") to each chunk containing a list for easier identification during retrieval.
5. **Test Retrieval Performance**:
   - Evaluate how well queries retrieve relevant chunks containing lists, ensuring no critical information is lost.

Example Workflow for Chunking Lists

1. Identify all list elements in the document using parsing tools (e.g., LangChain or Unstructured.io).
2. Group entire lists into single chunks alongside their lead-in text.
3. Apply overlap settings if splitting is necessary due to size constraints.
4. Use specialized chunkers for structured formats like Markdown or LaTeX to preserve formatting.

By applying these strategies, you can ensure that list items are handled effectively during chunking, preserving their structure and semantic relevance for downstream tasks like retrieval or analysis.

⁂

How can I handle list items that exceed the maximum chunk size

Handling list items that exceed the maximum chunk size involves strategies to preserve their structure and context while adhering to size constraints. Here are effective methods based on the search results and best practices:

Strategies for Handling Oversized List Items

1. Split the List into Smaller Chunks

- **Description**: Divide the oversized list into smaller sublists that fit within the maximum chunk size.
- **Implementation**:
  - Use a "maximally packed" approach to ensure each sublist is as large as possible without exceeding the size limit.
  - Maintain the order of items in the original list.
- **Example** (Python):

```
def split_list(lst, max_size):
    current_chunk = []
    current_size = 0
    for item in lst:
        item_size = len(item)  # Replace with token count if needed
        if current_size + item_size &gt; max_size:
            yield current_chunk
            current_chunk = []
            current_size = 0
        current_chunk.append(item)
        current_size += item_size
```

```
    if current_chunk:
        yield current_chunk

# Example usage
oversized_list = ["item1", "item2", "item3"]
chunks = list(split_list(oversized_list, max_size=10))
print(chunks)
```

- **Advantages**:
  - Preserves list structure and order.
  - Ensures no chunk exceeds the size constraint.

2. Use Overlap for Context Continuity

- **Description**: Add overlapping elements between chunks to maintain semantic continuity across splits.
- **Implementation**:
  - Introduce a configurable overlap parameter (e.g., `overlap=1`) to repeat one or more items from the previous chunk in the next chunk.
- **Example**:
  - Chunk 1: [Item1, Item2]
  - Chunk 2: [Item2, Item3]
- **Advantages**:
  - Reduces loss of context at chunk boundaries.

3. Combine Small List Items

- **Description**: If individual list items are smaller than the maximum chunk size, combine them with adjacent items until the chunk is maximally filled.
- **Implementation**:
  - Use a threshold (e.g., `combine_text_under_n_chars`) to merge small items into larger chunks.
- **Example**:
  - Combine "Item A" (5 tokens) and "Item B" (10 tokens) into a single chunk if the limit is 20 tokens.
- **Advantages**:
  - Prevents creating excessively small chunks, which can dilute retrieval relevance.

4. Isolate Large List Items

- **Description**: If a single list item exceeds the maximum chunk size, isolate it and split it further using text-splitting techniques.
- **Implementation**:
  - Use recursive splitting for oversized elements (e.g., split long sentences or paragraphs within the item).
- **Example**:
  - For a long bullet point, split it into sentences or phrases while retaining its meaning.
- **Advantages**:
  - Ensures large items are still processed while adhering to size constraints.

5. Preserve Hierarchy in Nested Lists

- **Description**: For nested lists, ensure that parent-child relationships are preserved within chunks.
- **Implementation**:
  - Group parent items with their corresponding child items, even if this requires splitting other parts of the list differently.
- **Example**:

```
- Parent Item
  - Child Item A
  - Child Item B
```

- **Advantages**:
  - Retains logical relationships between nested elements.

6. Use Metadata for Retrieval

- **Description**: Add metadata to each chunk to indicate its position within the original list and its relationship to other chunks.
- **Implementation**:
  - Include metadata like "Chunk ID," "Parent List ID," or "Position in List."
- **Example Metadata**:

```
{
    "chunk_id": "1",
    "list_id": "123",
    "position": "start"
}
```

- **Advantages**:

- Facilitates accurate reconstruction and retrieval of list items during downstream tasks.

Best Practices

1. **Preserve Contextual Integrity**: Ensure that list items retain their meaning by avoiding mid-item splits whenever possible.
2. **Optimize Chunk Size Dynamically**: Adjust chunk sizes dynamically based on item length and semantic importance.
3. **Validate Chunking Results**: Test retrieval performance using representative queries to ensure no critical information is lost due to splitting.

By applying these strategies, you can handle oversized list items effectively while maintaining their structure and relevance for downstream tasks like retrieval or analysis.

⁂

How do you build production grade document processing and indexing pipeline?

Building a production-grade document processing and indexing pipeline involves designing a robust architecture that ensures scalability, reliability, and efficiency. Below is a step-by-step guide based on best practices and tools:

1. Define Pipeline Goals and Requirements

- **Scope**: Identify the types of documents (e.g., PDFs, Word files) and the expected volume.
- **Objectives**: Define goals such as efficient retrieval, data enrichment, compliance with regulations, or integration with downstream systems.
- **Key Features**:
  - Scalability for handling large document repositories.
  - Monitoring and alerting for pipeline health.
  - Change management to adapt to evolving requirements.

2. Pipeline Architecture Design

A typical pipeline includes the following stages:

A. Ingestion

- Collect documents from various sources like APIs, databases, cloud storage (e.g., AWS S3, Azure Blob Storage), or local directories.
- Tools:

- **Snowflake Stages**: For document ingestion and metadata management[60] [61].
- **Apache NiFi or Talend**: For orchestrating ingestion workflows[62].

B. Preprocessing

- Validate document attributes (e.g., size, format) to ensure only processable files proceed further.
- Apply Optical Character Recognition (OCR) for scanned documents using tools like Amazon Textract or Azure Form Recognizer[63] [64].
- Normalize formats (e.g., converting PDFs to text) using converters like Coveo PDF Extractor[65].

C. Chunking

- Break documents into smaller chunks for indexing and retrieval.
- Strategies:
  - Use token-based chunking for compatibility with embedding models (e.g., OpenAI's text-embedding models)[66] [67].
  - Preserve semantic coherence by chunking based on paragraphs, sections, or logical boundaries.

D. Data Enrichment

- Extract structured data using Natural Language Processing (NLP) models to enrich content with metadata such as keywords, tags, and entity recognition[64].
- Tools:
  - Amazon Comprehend for entity extraction[63].
  - Azure AI Services for entity recognition skillsets[67].

E. Indexing

- Create indices to enable efficient search and retrieval.
- Methods:
  - **Vector-Based Indexing**: Use embeddings to represent documents as numerical vectors for similarity search. Suitable for semantic search applications[66] [67].
  - **Non-Vector Indexing**: Use traditional methods like BM25 for keyword matching[66] [68].
  - Hybrid indexing combines both approaches for flexibility[69].

3. Build the Pipeline

Step-by-Step Implementation

1. Ingestion Module

Set up a system to ingest documents into an internal stage:

```
CREATE OR REPLACE STAGE my_pdf_stage DIRECTORY = (ENABLE = TRUE);
CREATE STREAM my_pdf_stream ON STAGE my_pdf_stage;
ALTER STAGE my_pdf_stage REFRESH;
```

Use tools like Snowflake Streams or AWS Lambda for automated ingestion[60] [63].

2. Preprocessing Module

Preprocess documents by applying OCR and format normalization:

```
from azure.ai.formrecognizer import DocumentAnalysisClient
client = DocumentAnalysisClient(endpoint=endpoint, credential=credential)
result = client.begin_analyze_document("prebuilt-layout", document).result()
```

Validate attributes such as file size and page count before processing.

3. Chunking Module

Use token-based splitters to chunk documents:

```
from pathway.xpacks.llm.splitters import TokenCountSplitter
splitter = TokenCountSplitter(max_tokens=512)
chunks = splitter.split(document_text)
```

Ensure semantic coherence in chunks by preserving context across sections[66] [67].

4. Enrichment Module

Apply NLP models to extract metadata:

```
from azure.ai.textanalytics import TextAnalyticsClient
entities = TextAnalyticsClient(endpoint=endpoint).recognize_entities(document_text)
```

Tag documents with keywords, categories, or extracted entities.

5. Indexing Module

Create vector-based indices using embeddings:

```
from pathway.stdlib.indexing.nearest_neighbors import BruteForceKnnFactory
retriever_factory = BruteForceKnnFactory(embedder=OpenAIEmbedder(api_key="API_KEY"))
store = DocumentStore(docs=chunks, retriever_factory=retriever_factory)
```

Alternatively, use BM25 indexing for exact keyword matches.

4. Monitoring and Maintenance

- Implement logging and monitoring tools like AWS CloudWatch or Azure Monitor to track pipeline performance.
- Set up alerts for failures or anomalies in ingestion, processing, or indexing stages[62] [61].
- Regularly review search logs to identify gaps in indexing and improve query relevance[65].

5. Best Practices

1. **Start Small**: Begin with a single pipeline; scale up as data volume grows[65].
2. **Clean Indexes**: Avoid indexing irrelevant data to improve retrieval performance[65].
3. **Utilize Metadata**: Enrich documents with metadata tags for better categorization and filtering during search[69].
4. **Automate Updates**: Use scheduled tasks to process new or updated documents incrementally[63] [61].
5. **Test Retrieval**: Continuously evaluate retrieval accuracy using representative queries.

Example Use Case

For regulated industries:

1. Use Amazon Textract for OCR and extraction of structured data from forms.
2. Apply NLP models via Amazon Comprehend for entity recognition.
3. Store processed data in Elasticsearch with lineage tracking via DynamoDB[63].

For RAG solutions:

1. Use Azure AI Search to chunk content dynamically during indexing.
2. Embed chunks using OpenAI's text embeddings.
3. Query the index via semantic ranking integrated with Azure OpenAI services[67].

By following these steps and leveraging appropriate tools, you can build a scalable production-grade document processing and indexing pipeline tailored to your business needs.

⁂

How to handle graphs & charts in RAG

Handling graphs and charts in Retrieval-Augmented Generation (RAG) systems requires specialized techniques to ensure accurate interpretation, retrieval, and integration of visual data alongside textual information. Below are strategies and best practices for effectively managing graphs and charts in RAG workflows:

Challenges with Graphs and Charts in RAG

1. **Complex Visual Data**: Graphs and charts often contain intricate patterns, relationships, and numerical data that require precise interpretation.
2. **Multimodal Integration**: Combining visual elements with text-based retrieval systems can be challenging due to modality differences.
3. **Context Preservation**: Ensuring that graphs and charts are retrieved with relevant textual context is critical for accurate generation.

Strategies for Handling Graphs and Charts

1. Convert Visual Data into Embeddings

- Use vision models like CLIP, ResNet, or ViT to convert graphs and charts into numerical embeddings that capture their semantic meaning.
- Store these embeddings in a vector database (e.g., Milvus, FAISS) alongside text embeddings for unified retrieval.
- Example:
    - A query like "What trends are increasing?" can match a chart with an upward-sloping line by comparing the embedding of the query with the chart's embedding.

2. Multimodal Retrieval

- Integrate multimodal models (e.g., Flamingo, BLIP-2) to process both text and visual data simultaneously.
- Use cross-modal attention mechanisms to link textual queries with relevant visual data during retrieval.
- Example:
    - In a financial RAG system, retrieve charts showing quarterly revenue trends alongside explanatory text.

3. Metadata Tagging

- Enrich graphs and charts with metadata during preprocessing (e.g., titles, axis labels, captions).
- Use this metadata to improve indexing and retrieval accuracy by associating visual elements with their textual descriptions.
- Example:
  - Tag a bar chart titled "Sales Growth 2024" with keywords like "sales," "growth," and "2024."

4. Hybrid Search

- Combine text-based search (e.g., Elasticsearch) with vector-based search for images to balance precision and recall.
- Use hybrid multivector search techniques to retrieve documents containing both relevant text and visual elements.
- Example:
  - For a query like "Show me the revenue comparison between 2023 and 2024," retrieve both the textual analysis and the corresponding bar chart.

5. Visual Parsing

- Extract structured data from graphs or charts using vision-language models (VLMs) or OCR tools like Azure Form Recognizer or Amazon Textract.
- Parse elements such as axis labels, legends, and data points into machine-readable formats for indexing alongside text data.
- Example:
  - Convert a pie chart into tabular data showing percentages for each category.

6. Chunking Visual Data

- Treat graphs and charts as distinct chunks during document processing.
- Include accompanying text (e.g., captions or explanations) in the same chunk to preserve context.
- Example:
  - Chunk a graph on "Market Share by Region" along with its caption explaining regional differences.

7. Multimodal Prompting

- Pass both textual content and image URIs to multimodal LLMs during query resolution.
- Design prompts that instruct the model to interpret visual elements accurately.
- Example:

```
"Analyze the trends shown in the bar chart titled 'Sales Growth' on page 5 of the rep
```

8. Precompute Insights

- Precompute insights from graphs during preprocessing using statistical analysis or AI models (e.g., GPT-based reasoning).
- Store these insights as additional metadata or summaries for faster retrieval during queries.
- Example:
  - For a line graph showing temperature trends, precompute insights like "Temperatures increased steadily over the past decade."

Tools for Managing Graphs and Charts in RAG

Tool	Use Case	Features
CLIP	Image-to-text alignment	Embeds images and text into a shared semantic space
BLIP-2	Multimodal attention	Combines vision-language features for complex queries
Azure Form Recognizer	Extracting structured data from visuals	Parses charts, tables, and diagrams
Milvus	Large-scale vector search	Stores embeddings for both text and images
Pathway MM-RAG	Multimodal RAG workflows	Handles PDFs with text, images, tables, and charts

Example Workflow for Graphs & Charts in RAG

1. **Preprocessing**:
   - Extract graphs/charts from documents using layout-aware parsers (e.g., PyMuPDF).
   - Generate embeddings using CLIP or ResNet models.
   - Tag visual elements with metadata (e.g., titles, captions).
2. **Indexing**:
   - Store embeddings in a vector database alongside text embeddings.
   - Use hybrid indexing techniques to enable multimodal retrieval.
3. **Query Resolution**:

- Convert user queries into embeddings using multimodal models.
- Retrieve relevant chunks containing both text and visual data.
- Pass retrieved content to an LLM for response generation.

4. **Generation**:
   - Use multimodal LLMs (e.g., GPT-4o) to interpret retrieved graphs/charts alongside textual context.
   - Generate responses grounded in both visual and textual information.

Practical Applications

1. **Finance**: Analyze revenue trends using bar charts alongside explanatory text from reports.
2. **Healthcare**: Interpret medical imaging (e.g., X-rays) along with patient notes for diagnosis queries.
3. **Scientific Research**: Retrieve experimental results stored as graphs along with their descriptions.

By leveraging these strategies, you can build robust RAG systems capable of handling graphs and charts effectively while ensuring accurate retrieval and generation of contextually rich responses.

⁂

1. https://www.couchbase.com/blog/data-chunking/
2. https://shelf.io/blog/demystifying-content-chunking-in-ai-and-enterprise-knowledge-management/
3. https://www.linkedin.com/pulse/ultimate-guide-chunking-mastering-data-processing-analysis-jaleel-danbc
4. https://www.datastax.com/blog/chunking-to-get-your-data-ai-ready
5. https://www.pinecone.io/learn/chunking-strategies/
6. https://unstructured.io/blog/chunking-for-rag-best-practices
7. https://www.nb-data.com/p/9-chunking-strategis-to-improve-rag
8. https://esipfed.github.io/cloud-computing-cluster/optimization-practices.html
9. https://www.metriccoders.com/post/what-factors-influence-chunk-size
10. https://www.pinecone.io/learn/chunking-strategies/
11. https://www.datastax.com/blog/chunking-to-get-your-data-ai-ready
12. https://www.galileo.ai/blog/mastering-rag-advanced-chunking-techniques-for-llm-applications
13. https://unstructured.io/blog/chunking-for-rag-best-practices
14. https://www.linkedin.com/pulse/chunking-best-practices-retrieval-augmented-generation-rishabh-goyal-hol3c
15. https://www.llamaindex.ai/blog/evaluating-the-ideal-chunk-size-for-a-rag-system-using-llamaindex-6207e5d3fec5
16. https://vectorize.io/evaluating-the-ideal-chunk-size-for-a-rag-system/

17. https://weaviate.io/developers/academy/py/standalone/chunking/considerations
18. https://www.pinecone.io/learn/chunking-strategies/
19. https://www.nb-data.com/p/9-chunking-strategis-to-improve-rag
20. https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-chunk-documents
21. https://successive.tech/blog/rag-models-optimizing-text-input-chunking-splitting-strategies/
22. https://www.f22labs.com/blogs/7-chunking-strategies-in-rag-you-need-to-know/
23. https://www.rackspace.com/blog/how-chunking-strategies-work-nlp
24. https://www.sagacify.com/news/a-guide-to-chunking-strategies-for-retrieval-augmented-generation-rag
25. https://www.mongodb.com/developer/products/atlas/choosing-chunking-strategy-rag/
26. https://unstructured.io/blog/chunking-for-rag-best-practices
27. https://www.couchbase.com/blog/data-chunking/
28. https://www.pinecone.io/learn/chunking-strategies/
29. https://www.couchbase.com/blog/data-chunking/
30. https://vectorize.io/evaluating-the-ideal-chunk-size-for-a-rag-system/
31. https://www.mongodb.com/developer/products/atlas/choosing-chunking-strategy-rag/
32. https://www.reddit.com/r/LangChain/comments/16bjj6w/what_is_optimal_chunk_size/
33. https://www.llamaindex.ai/blog/evaluating-the-ideal-chunk-size-for-a-rag-system-using-llamaindex-6207e5d3fec5
34. https://datasciencedojo.com/blog/rag-application-with-llamaindex/
35. https://www.linkedin.com/pulse/my-basic-guide-understanding-chunking-generative-ai-akash-pandey-3asee
36. https://research.aimultiple.com/digitization-best-practices/
37. https://fractal.ai/the-power-of-document-digitization/
38. https://arxiv.org/html/2402.05131v2
39. https://www.f22labs.com/blogs/7-chunking-strategies-in-rag-you-need-to-know/
40. https://www.snowflake.com/en/engineering-blog/impact-retrieval-chunking-finance-rag/
41. https://www.nb-data.com/p/9-chunking-strategis-to-improve-rag
42. https://unstructured.io/blog/chunking-for-rag-best-practices
43. https://www.ragie.ai/blog/our-approach-to-table-chunking
44. https://docs.unstructured.io/api-reference/api-services/chunking
45. https://cloud.google.com/agentspace/agentspace-enterprise/docs/parse-chunk-documents
46. https://blog.gopenai.com/chunking-pdfs-and-multimodal-documents-efficient-methods-for-handling-text-tables-and-images-for-467472f02d34
47. https://docs.unstructured.io/api-reference/partition/chunking
48. https://www.ragie.ai/blog/our-approach-to-table-chunking
49. https://github.com/Unstructured-IO/unstructured/issues/3778
50. https://www.reddit.com/r/LangChain/comments/16uip55/chunking_and_retrieving_documents_with_tables/
51. https://www.mongodb.com/developer/products/atlas/choosing-chunking-strategy-rag/

52. https://techcommunity.microsoft.com/t5/datacat/top-10-best-practices-for-building-a-large-scale-relational-data/ba-p/305158
53. https://accreditly.io/articles/how-to-manage-large-table-to-keep-them-optimized
54. https://builtin.com/articles/optimize-sql-for-large-data-sets
55. https://planetscale.com/blog/dealing-with-large-tables
56. https://www.site24x7.com/learn/optimize-slow-sql-queries-for-large-dataset.html
57. https://www.acceldata.io/blog/sql-performance-tuning-strategies-to-optimize-query-execution
58. https://www.turing.com/resources/best-big-data-platforms
59. https://fiveable.me/lists/major-big-data-frameworks
60. https://docs.snowflake.com/en/user-guide/snowflake-cortex/document-ai/tutorials/create-processing-pipelines
61. https://quickstarts.snowflake.com/guide/doc-ai-pipeline-automation/index.html
62. https://www.secoda.co/glossary/what-are-production-grade-data-pipelines
63. https://github.com/aws-samples/document-processing-pipeline-for-regulated-industries
64. https://learn.microsoft.com/en-us/azure/architecture/ai-ml/architecture/automate-document-processing-azure-form-recognizer
65. https://www.coveo.com/blog/indexing-pipelines/
66. https://pathway.com/developers/user-guide/llm-xpack/docs-indexing/
67. https://learn.microsoft.com/en-us/azure/search/tutorial-rag-build-solution-pipeline
68. https://teamhub.com/blog/a-comprehensive-guide-to-document-indexing/
69. https://arya.ai/blog/document-indexing