Chunking Considerations

Several variables play a role in determining the best chunking strategy, and these variables vary depending on the use case. Here are some key aspects to keep in mind:

1. **What is the nature of the content being indexed?** Are you working with long documents, such as articles or books, or shorter content, like tweets or instant messages? The answer would dictate both which model would be more suitable for your goal and, consequently, what chunking strategy to apply.
2. **Which embedding model are you using, and what chunk sizes does it perform optimally on?** For instance, sentence-transformer models work well on individual sentences, but a model like text-embedding-ada-002 performs better on chunks containing 256 or 512 tokens.
3. **What are your expectations for the length and complexity of user queries?** Will they be short and specific or long and complex? This may inform the way you choose to chunk your content as well so that there's a closer correlation between the embedded query and embedded chunks.
4. **How will the retrieved results be utilized within your specific application?** For example, will they be used for semantic search, question answering, summarization, or other purposes? For example, if your results need to be fed into another LLM with a token limit, you'll have to take that into consideration and limit the size of the chunks based on the number of chunks you'd like to fit into the request to the LLM.

Latest Trends in Chunking Strategies

- **Anthropic's contextual embedding:** Contextual retrieval strategy which adds chunk-specific explanatory context to each chunk before embedding ("Contextual Embeddings") and creating the BM25 index ("Contextual BM25"). This contextual chunk addition is facilitated by an LLM. To implement contextual retrieval, instructions are given to the LLM model to provide concise chunk-specific context that explains the chunk using the context of the overall document.[7]\
- **ChunkRAG:** A framework that improves RAG systems by evaluating and filtering retrieved information at the chunk level. This approach uses semantic chunking to split documents into meaningful sections and

applies LLM-based relevance scoring to determine each chunk's alignment with the user's query.[8]

Impact of Different Combinations

chunk_size	chunk_overlap	Impact
Large	Small	Fewer, larger chunks with minimal redundancy. May lose context at boundaries.
Large	Large	Fewer chunks with significant overlap, maintaining context but increasing redundancy.
Small	Small	Many small, independent chunks. Fast but context loss at boundaries is likely.
Small	Large	Many small chunks with overlapping context, preserving information but increasing cost.

Why Recursive Chunking Was Chosen Over Semantic Chunking

1. **Efficiency**:
   - **Recursive Chunking**: Splits text based on fixed character sizes with overlap (e.g., 1000 characters with 100-character overlap). This approach is computationally efficient as it avoids heavy NLP processing.
   - **Semantic Chunking**: Requires parsing the text using advanced NLP techniques to identify logical breakpoints (e.g., topic shifts, sentence boundaries). This increases computation time and latency.

   Reason: Recursive chunking was chosen for its faster processing speed, ensuring the system remains efficient for real-time applications.

2. **Scalability**:
   - **Recursive Chunking**: Scales well with large datasets, as it uses a straightforward algorithm that works across diverse document types without additional overhead.
   - **Semantic Chunking**: Introduces complexity and higher resource usage, especially for large or unstructured datasets, making it less scalable.

   Reason: Recursive chunking scales better when processing large volumes of documents in a time-sensitive environment.

3. **Implementation Simplicity**:
   - **Recursive Chunking**: Easy to implement and integrates seamlessly with retrieval pipelines and embedding models.
   - **Semantic Chunking**: Requires additional pre-processing steps, such as running NLP models or topic modeling, increasing system complexity.

Reason: Recursive chunking provides a simpler yet effective approach, reducing development and maintenance efforts.

4. **Embedding Quality**:
    - **Recursive Chunking**: Ensures chunks retain logical flow with overlapping regions, allowing embedding models (e.g., OllamaEmbeddings) to generate accurate vector representations.
    - **Semantic Chunking**: Produces highly coherent chunks but risks uneven sizes or excessive truncation in embedding windows if topics vary greatly in length.

    Reason: Recursive chunking balances coherence and uniformity, ensuring consistent embedding quality without introducing variability in chunk sizes.

5. **Performance on Document Diversity**:
    - **Recursive Chunking**: Works uniformly across structured (e.g., reports, research papers) and unstructured (e.g., raw text) documents.
    - **Semantic Chunking**: May struggle with unstructured or highly variable text, where semantic boundaries are ambiguous or hard to detect.

    Reason: Recursive chunking performs robustly across a variety of document types, making it more versatile for the project.

Embedding models are models that are trained specifically to generate *vector embeddings*: long arrays of numbers that represent semantic meaning for a given sequence of text:

https://www.nomic.ai/blog/posts/nomic-embed-text-v1

Why Nomic Text Embedding Is a Strong Choice

1. **Long Context-Length (8192 Tokens)**
    - **Advantage**: Supports a much longer context length compared to many open-source alternatives, matching OpenAI's Ada-002 model.
    - **Why It Matters**: Ideal for handling large documents or combining multiple chunks into a single representation, reducing the need for excessive chunking and improving performance on long-context tasks.
2. **Outperforms Key Competitors**
    - **Compared to OpenAI's Ada-002**: Matches or exceeds performance on both short and long-context tasks, offering an open-source alternative with comparable results.
    - **Compared to Open-Source Models (e.g., E5-Mistral)**: Achieves superior performance while maintaining practical model size and inference speed, unlike other long-context models that are too large or inefficient.
3. **Open-Source and Auditable**
    - **Advantage**: Released under the Apache-2 license, making it open for modification, deployment, and adaptation without licensing costs.

    - **Why It Matters**: Training data and methodologies are fully transparent, unlike closed-source models like Ada-002, ensuring reliability and trustworthiness for production applications.
4. **Broad Applications**
    - **RAG for LLMs**: Powers retrieval-augmented generation pipelines by encoding meaningful semantic representations for both short and long queries.
    - **Semantic Search**: Excels at creating embeddings that enhance precision in information retrieval tasks.
    - **Clustering and Classification**: Enables high-quality data visualization, topic modeling, and document categorization with robust embeddings.
5. **Efficient and Enterprise-Ready**
    - **Free Tokens for Production Use**: Nomic Embed includes 1M free tokens for its embedding API, allowing cost-effective experimentation and deployment.
    - **Enterprise Integration**: Provides a secure and compliant enterprise-grade solution, ensuring scalability and privacy for organizations.
6. **Transparency in Training and Weights**
    - **Advantage**: Unlike Ada-002, Nomic provides both model weights and training code, enabling users to fine-tune or audit the model for specific use cases.
    - **Why It Matters**: Organizations can adapt the model for domain-specific needs without relying on proprietary, opaque systems.
7. **Competitive Performance on General and Long-Context Tasks**
    - **Why It Excels**: Balances strong performance with practical usability, unlike alternatives like E5-Mistral that may be impractical for deployment due to large size or slower inference speeds.

https://www.bentoml.com/blog/a-guide-to-open-source-embedding-models

Key Modifications to Standard BERT

1. **Rotary Position Embeddings (RoPE)**:
    - **Why**: Standard BERT uses absolute positional embeddings, which limit its ability to extrapolate beyond its trained context length (512 tokens).
    - **Change**: Replace absolute positional embeddings with **Rotary Position Embeddings**. RoPE allows extrapolation to longer context lengths by encoding positional information in a manner that generalizes better for sequences longer than what the model was trained on.
2. **SwiGLU Activations**:
    - **Why**: BERT originally uses GELU activations, but **SwiGLU** (a variation of gated linear units) has been shown to improve model performance in terms of efficiency and expressiveness.
    - **Change**: SwiGLU is used in the feed-forward layers of the transformer blocks.
3. **Dropout Set to 0**:
    - **Why**: Dropout is typically used for regularization to prevent overfitting, but modern optimizations like large batch sizes and precision improvements reduce the need for it.
    - **Change**: Dropout is completely removed to improve training efficiency and speed.

Training Optimizations

1. **DeepSpeed and FlashAttention**:
    - **Why**: Training models with a context length of 2048 is computationally expensive. These tools improve efficiency:
        - **DeepSpeed**: A library for distributed training, allowing efficient scaling across multiple GPUs.
        - **FlashAttention**: Optimizes the computation of the attention mechanism, reducing memory usage and increasing speed.
2. **BF16 Precision**:
    - **Why**: Instead of using full precision (FP32), **BF16 (bfloat16)** is used to reduce memory usage and speed up training without significantly affecting model accuracy.
3. **Increased Vocabulary Size**:
    - **Why**: Standard BERT has a vocabulary size of 30,000 tokens, which is not divisible by hardware-friendly sizes. Increasing the vocabulary size to a multiple of 64 (e.g., 32,000) improves GPU memory alignment and efficiency.
4. **Larger Batch Size**:
    - **Why**: A batch size of 4096 allows for faster convergence and better utilization of hardware.
5. **Higher Masking Rate**:
    - **Why**: During Masked Language Modeling (MLM), the masking rate is increased from 15% (standard in BERT) to **30%**. This makes the model more robust by encouraging it to predict masked tokens in a wider variety of contexts.
6. **No Next Sentence Prediction (NSP)**:
    - **Why**: The NSP objective in BERT has been found to contribute little to downstream task performance. Removing it simplifies training and reduces unnecessary overhead.

Evaluation

- **Metric**: The model is evaluated on the **GLUE benchmark**, a collection of tasks for natural language understanding.
- **Performance**:
    - Performs **comparably** to standard BERT models on these tasks.
    - **Advantage**: Supports a **context length of 2048 tokens**, which makes it much better for tasks involving long documents or large contexts.

Feature	FAISS	Weaviate	Qdrant	Milvus	Pinecone
Open Source	Yes	Yes	Yes	Yes	No (Proprietary)
Scalability	Moderate	High (distributed with Kubernetes)	High (real-time vector search)	Very High (distributed architecture)	Very High (fully managed)
Performance	Extremely fast for similarity search	Moderate (depends on scale & metadata)	High-performance real-time queries	High-performance at enterprise scale	High-performance with managed infra
Metadata Support	Limited	Strong filtering support	Advanced filtering and metadata search	Advanced filtering and indexing options	Strong filtering and metadata handling
Ease of Use	Requires some setup	Easy to set up with pre-built modules	Easy to integrate with Python APIs	Requires expertise for deployment	Easiest (fully managed service)
Cost	Free (open source, local use)	Free (self-hosted)	Free (self-hosted)	Free (self-hosted)	Paid (based on API usage)
Best Use Case	Local systems for high-speed retrieval tasks	Metadata-rich retrieval & hybrid search	Real-time applications with metadata	Large-scale enterprise search systems	Cloud-based RAG pipelines

Summary of Decision

Criteria	IndexFlatL2	HNSW	Reason for Choice
Accuracy	Exact (guaranteed nearest neighbors)	Approximate (potentially less accurate)	Accuracy is critical for high-quality response generation.
Dataset Size	Ideal for small to medium datasets	Optimized for large datasets	Dataset size is manageable, so exact search is feasible.
Latency	Moderate (~50-100ms/query)	Low (~10-50ms/query) for large datasets	Latency difference is negligible for our dataset size.
Memory Usage	Low (stores raw vectors only)	High (graph structure overhead)	Resource constraints favor a lightweight solution.
Implementation Complexity	Simple and straightforward	Requires graph construction and tuning	Simplicity aligns with the project's rapid development goals.

- **Parameters:**
  - `search_type="mmr"`:
    - Ensures that the retrieved results are not only relevant to the query but also diverse.
  - `k=3`:
    - Returns the top 3 results.
  - `fetch_k=100`:
    - Considers the top 100 candidates before selecting the 3 results, ensuring better diversity.
  - `lambda_mult=1`:
    - Balances relevance and diversity:
      - **λ=1**: Focuses entirely on relevance.
      - **λ<1**: Incorporates diversity into the results.

1. General Parameters

Key	Description	Example Value
`k`	The number of top results to return.	`3` (returns top 3 results)
`fetch_k`	The number of initial candidates to consider before selecting the top `k` results (useful for MMR).	`100`
`filter`	A filter function or dictionary to apply metadata-based constraints on the results.	`{"source": "pdf"}`

2. Maximal Marginal Relevance (MMR) Parameters

Key	Description	Example Value
`lambda_mult`	Weighting factor for balancing relevance vs. diversity in MMR.	`1` (relevance-focused), `0.5` (balanced)
`similarity_metric`	The similarity metric to use for comparison (e.g., `"cosine"`, `"euclidean"`).	`"cosine"`

3. Similarity Search Parameters

Key	Description	Example Value
`threshold`	Minimum similarity score for a result to be included in the retrieval.	`0.8`
`distance_metric`	Distance function for vector comparison.	`"l2"` or `"cosine"`

Search Type	When to Use
"mmr"	When you want a balance of relevance and diversity in the retrieved results.
"similarity"	For basic retrieval tasks where relevance is the only concern.
"hybrid"	When combining semantic search with keyword-based matching improves results.
"maximal_inner_product"	For recommendation systems or normalized embedding spaces.
"approximate"	For large datasets where retrieval speed is critical.
"exact"	For small datasets where precision is more important than speed.
"bm25"	For keyword-focused search tasks.

Comparison of Search Types

Search Type	Metric	Key Use Case	Strengths	Weaknesses
Cosine Similarity	Angle between vectors	NLP embeddings, normalized vectors	Works well with most embeddings.	Assumes vectors are normalized.
MMR	Relevance & diversity	Broad queries, diverse contexts	Reduces redundancy, improves diversity.	Requires tuning (e.g., `lambda_mult`).
Euclidean Distance	Straight-line distance	High-dimensional spaces	Simple to compute.	Sensitive to vector magnitude.
Dot Product	Alignment of vectors	Recommendation systems, normalized vectors	Fast and straightforward.	Assumes normalized vectors.
Hybrid Search	Multi-method	Metadata-rich or complex queries	Combines strengths of methods.	Computationally intensive.
Keyword Search	Text matching	Simple queries, fast retrieval	No preprocessing needed.	Lacks semantic understanding.

Retrieval Method	Description	Use Case
Vector Store Flat Index	Stores data in vectorized form and uses similarity metrics (e.g., cosine similarity) to match query vectors with content vectors.	Simple, effective similarity search for embedding-based retrieval.
Hierarchical Indices	Uses a two-step approach: retrieves summaries first, then performs a detailed search within selected document chunks.	Efficient for large datasets by reducing the search space.
Hypothetical Questions	Uses LLMs to generate hypothetical questions or responses for text chunks, and stores their embeddings for semantic similarity matching.	Improves search quality by focusing on semantically relevant chunks.
HyDE (Hypothetical Answer)	Generates hypothetical answers based on a query, then uses the generated vector for a refined search.	Enhances query precision and alignment with context.
Small-to-Big Retrieval	Links smaller chunks to parent chunks (e.g., sentences to paragraphs). Retrieves larger chunks once a smaller one is matched.	Provides broader context for LLMs by retrieving surrounding information.
Fusion Retrieval	Combines multiple retrieval methods (e.g., vector search + keyword search) to enhance result accuracy.	Increases relevance and mitigates limitations of individual retrieval methods.
Reranking and Filtering	Reorders and filters retrieved results using machine learning or heuristic models to prioritize relevance.	Ensures high-quality inputs for LLMs after the initial retrieval step.
Query Transformation	Reformulates the original query (e.g., expansion, paraphrasing) to better capture intent and improve retrieval performa.	Optimizes query-document matching by aligning the query with the indexed data.

Comparison Table

Model	Type	Size	Inference Speed	Generative Quality	Context Handling	Resource Usage	Best Use Case
LLaMA 3.2:1B	LLM	~1 Billion	~200-300ms	High	Strong	Moderate (GPU required)	Medium-complexity RAG tasks where quality and contextual grounding are critical.
Mistral 7B	LLM	~7 Billion	~200ms	High	Strong	Moderate-High (GPU needed)	High-complexity RAG tasks with moderately long contexts and need for local deployment.
Phi-3 Mini	SLM	~3 Billion	~100-150ms	Moderate	Moderate	Moderate (GPU or CPU)	Cost-effective RAG pipelines with moderate query complexity and real-time response

Model	Type	Size	Inference Speed	Generative Quality	Context Handling	Resource Usage	Best Use Case
							requirements.
TinyLlama	SLM	<1 Billion	~50-100ms	Low	Weak	Very Low (CPU-only)	Extremely lightweight tasks with minimal hardware requirements and very simple queries.
GPT-4	LLM	>175 Billion	~500-1000ms (API)	Very High	Very Strong	Very High (API-dependent)	High-complexity RAG tasks requiring deep contextual understanding and nuanced generative outputs.
Claude 2	LLM	~50 Billion	~300ms (API)	High	Very Strong	High (API-dependent)	Long-context RAG tasks with safety-critical or alignment-focused responses.
DistilBERT	SLM	~66 Million	~50ms	Moderate	Weak	Very Low	Lightweight QA or embedding generation for basic RAG workflows.
T5-Small	SLM	~60 Million	~100ms	Moderate	Weak	Low	Text-to-text tasks like summarization or simple question answering.
GPT-J (6B)	LLM	~6 Billion	~200ms	Moderate	Moderate	High	Lightweight open-source RAG tasks with a need for decent generative performance.

Analysis by Key Features

1. Inference Speed

- **Fastest Models**:
  - **TinyLlama** (~50-100ms) is the fastest, suitable for low-latency applications.
  - **Phi-3 Mini** (~100-150ms) balances speed and generative quality.
- **Slowest Models**:
  - **GPT-4** (~500-1000ms via API) and **Claude 2** (~300ms via API) are slower but offer unparalleled generative quality and contextual understanding.

2. Generative Quality

- **Top Performers**:

    - **GPT-4** leads in quality, capable of nuanced and complex answers.
    - **Mistral 7B** and **Claude 2** perform very well for high-context RAG tasks.
- **Lower Performers**:
    - **TinyLlama** and **DistilBERT** struggle with generative complexity and contextual coherence.

3. Context Handling

- **Best Context Handling**:
    - **GPT-4**, **Claude 2**, and **Mistral 7B** excel with strong attention mechanisms and longer context windows.
- **Limited Context Handling**:
    - **TinyLlama** and **DistilBERT** are limited to shorter contexts, reducing their applicability in complex multi-turn RAG tasks.

4. Resource Usage

- **Efficient Models**:
    - **Phi-3 Mini** and **TinyLlama** are resource-efficient, suitable for lightweight setups (even CPU-based for TinyLlama).
- **Resource-Intensive Models**:
    - **GPT-4** and **Mistral 7B** require high-end GPUs or API access, increasing deployment costs.

5. Use Case Suitability

- **Real-Time Applications**:
    - **Phi-3 Mini** and **TinyLlama** are ideal for tasks requiring fast response times and minimal hardware.
- **Complex RAG Tasks**:
    - **GPT-4** and **Claude 2** are suited for nuanced, domain-specific queries where accuracy and depth are paramount.
- **Intermediate RAG Tasks**:
    - **LLaMA 3.2:1B** and **Mistral 7B** offer a balance between quality and efficiency, making them versatile options.

Which Model to Choose for Your Task?

If Speed and Efficiency Are Priorities:

- **Phi-3 Mini**: Offers a balance of speed and moderate generative quality. Ideal for mid-complexity RAG pipelines with real-time performance requirements.
- **TinyLlama**: Best for simple RAG tasks with low hardware capabilities.

If Quality and Context Handling Are Critical:

- **GPT-4**: Best for intricate, long-context queries requiring deep reasoning and aligned outputs.
- **Mistral 7B**: Suitable for tasks requiring strong generative performance with moderate latency and resource requirements.

- **LLaMA 3.2:1B**: A good trade-off between performance and resource usage, ideal for mid-level RAG systems.

If Cost Is a Concern:

- **Phi-3 Mini**: Cost-effective while delivering moderate generative quality.
- **DistilBERT**: Suitable for basic QA or embedding generation with minimal cost.

Knowledge Graph Embeddings: A Clear Overview

Knowledge Graph Embeddings (KGEs) are techniques that represent entities (nodes) and relationships (edges) in a **knowledge graph (KG)** as continuous vectors in a low-dimensional space. These embeddings make KGs computationally efficient for machine learning tasks like **link prediction**, **classification**, and **semantic similarity**.

What is a Knowledge Graph?

A knowledge graph consists of:

1. **Entities (Nodes)**: Represent objects or concepts (e.g., "Albert Einstein," "Germany").
2. **Relations (Edges)**: Define the relationships between entities (e.g., "born_in," "lives_in").

For example:

- Triple: **(Albert_Einstein, born_in, Germany)**

Why Use Knowledge Graph Embeddings?

1. **Efficient Representation**: Convert high-dimensional graph structures into dense, low-dimensional vectors.
2. **Machine Learning Ready**: Enable KGs to be used in machine learning models for tasks like classification or clustering.
3. **Semantic Insights**: Encodes the relationships and semantics of the graph in the vector space.

How Knowledge Graph Embeddings Work

The goal is to represent:

- **Entities** as vectors: Entity: Ei∈Rd\text{Entity: } \mathbf{E}_i \in \mathbb{R}^dEntity: Ei∈Rd
- **Relations** as transformations: Relation: Ri∈Rd\text{Relation: } \mathbf{R}_i \in \mathbb{R}^dRelation: Ri∈Rd

For example:

- For the triple (h,r,t)(h, r, t)(h,r,t), embeddings ensure Eh+Rr≈Et\mathbf{E}_h + \mathbf{R}_r \approx \mathbf{E}_tEh+Rr≈Et.

Techniques for Knowledge Graph Embeddings

1. Translational Models

- Represent relations as translations in vector space.
- **Popular Methods**:
  - **TransE**: Models relations as vector translations (h+r≈th + r \approx th+r≈t).
  - **TransH**: Projects entities onto relation-specific hyperplanes for better flexibility.
  - **TransR**: Embeds entities and relations in separate spaces for complex graphs.

2. Matrix Factorization Models

- Factorize the adjacency or tensor representation of the graph.
- **Popular Methods**:
  - **DistMult**: Uses a diagonal matrix for relations, focusing on symmetric relationships.
  - **ComplEx**: Extends DistMult to handle asymmetric relations by using complex vectors.

3. Deep Learning Models

- Use neural networks to capture non-linear patterns in KGs.
- **Popular Methods**:
  - **ConvE**: Applies convolutional layers to entity-relation pairs for better feature extraction.
  - **R-GCN**: Extends graph convolutional networks to include relational information.

4. Random Walk-Based Models

- Learn embeddings by sampling sequences from the graph.
- **Popular Methods**:
  - **DeepWalk**: Generates embeddings by applying skip-gram to random walk sequences.
  - **Node2Vec**: Improves DeepWalk by introducing biased random walks to capture graph structure.

5. Attention-Based Models

- Use attention mechanisms to weigh the importance of nodes and edges.
- **Popular Methods**:
  - **KBGAT (Knowledge Graph Attention Network)**: Captures the relevance of neighbors dynamically.

Applications of Knowledge Graph Embeddings

1. Link Prediction

- Predicts missing edges in a graph.
- Example: From **(Albert_Einstein, ?, Germany)**, predict \text{"born_in"}.

2. Entity Classification

- Classifies entities into categories.
- Example: Classify "Albert Einstein" as a "Scientist."

3. Relation Extraction

- Extracts relationships between entities.
- Example: Identify **"lives_in"** from **(Marie_Curie, ?, Paris)**.

4. Semantic Similarity

- Measures the similarity between entities.
- Example: Compute similarity between **"Physics"** and **"Science."**

5. Question Answering

- Enables KG-based systems to answer natural language queries.
- Example: "Where was Albert Einstein born?" → **"Germany."**

Popular Evaluation Metrics

1. **Mean Reciprocal Rank (MRR)**: Evaluates ranking quality of predictions.
2. **Hits@k**: Measures how often the correct answer is in the top-k results.
3. **Mean Rank**: Average rank of the correct prediction (lower is better).

Popular Frameworks for Knowledge Graph Embeddings

1. **PyKEEN**:
    - Comprehensive library for knowledge graph embedding algorithms.
2. **DGL-KE**:
    - Scalable embeddings using the Deep Graph Library.
3. **OpenKE**:
    - Framework for KG embedding training and evaluation.
4. **AmpliGraph**:
    - Supports link prediction