RAG と BM25 解説書 アウトライン

検索拡張生成（RAG: Retrieval-Augmented Generation）において、高精度な回答を生成するには的確なドキュメントを検索する仕組みが欠かせません。本書では、意味の近さで探す「ベクトル検索」と、正確な単語の一致で探す伝統的な全文検索アルゴリズム「BM25」を組み合わせたハイブリッド検索について解説します。さらに、Python と LangChain による実践的な実装方法、主要データストアの特徴と選び方、そして実プロジェクトでの大規模データ（235万件）における BM25 アーキテクチャとハイブリッド運用評価までを網羅します。

章構成

- **1. RAGと情報検索の基本**

  RAG の全体像と検索（Retrieval）の重要性、LLM 単体での限界（ハルシネーションや知識の古さ）を解消する仕組み、検索手法の分類を学びます。

- **2. BM25アルゴリズムの仕組み**

  BM25（Best Matching 25）の背景、TF-IDF との違い、計算式の意味（単語頻度の飽和、文書長によるペナルティ）、日本語における形態素解析の役割を理解します。

- **3. ベクトル検索とBM25の比較**

  密ベクトル検索（Dense Retrieval）と疎検索（Sparse Retrieval: BM25）のメリット・デメリットを対比し、なぜ両者を併用するハイブリッド検索が必要なのかを解説します。

- **4. 主要データストアの比較と選定**

  RAG で利用される主要なデータストア（Chroma、FAISS、Qdrant、Milvus、Pinecone、Elasticsearch / OpenSearch、PostgreSQL / pgvector）の特徴、メリット・デメリット、選定基準をまとめます。

- **5. LangChainによるBM25の実装**

  Python と LangChain（langchain-community）を使用して、日本語テキストを適切にトークナイズした BM25Retriever を構築し、キーワード検索を実行する手順を学びます。

- **6. ハイブリッド検索とEnsembleRetriever**

  Ollama（bge-m3）によるベクトル検索と BM25 検索を統合する EnsembleRetriever の実装方法、Reciprocal Rank Fusion（RRF）のアルゴリズム、重みパラメータの調整法を解説します。

- **7. リランキングとRAGパイプラインの構築**

  検索結果の順位を再評価する Cross-Encoder（リランカー）の適用手法と、検索した文書を Ollama（gemma4:12b）に渡して回答を生成する完全なローカル RAG パイプラインを LCEL で構築します。

- **8. 実践データソース別の索引システム構築**

  IRIS バンドルデータ（構造化 Markdown）、LOTUS ノートデータ（差分更新）、大容量テキストデータ（行単位ストリーミング）、複数 PDF ドキュメントに対する具体的な前処理・チャンキング手順と、BM25 およびベクトルインデックスの永続化手法を学びます。

- **9. 実践事例：英辞郎データ（235万件）の高速BM25検索アーキテクチャ**

  約 160MB・235 万件の英辞郎辞書テキストを対象に、Pure Python のインメモリ保持限界を克服し、SQLite FTS5 と文字バイグラムによるゼロ外部依存・メモリ 40MB・応答 1 〜 3ms の高速 BM25 検索基盤を構築した実プロジェクトの設計とトラブルシューティングを解説します。

- **10. 実践評価：大規模辞書におけるベクトル検索とBM25のハイブリッド運用設計**

  235 万件規模の大規模データにおける全件ベクトル化のコストボトルネック（Embedding 推論時間、ストレージ・メモリ消費、入力補完レイテンシ）を評価し、BM25・局所ベクトル DB・ローカル LLM を適材適所で組み合わせる「3層ハイブリッド戦略」を解説します。

- **11. まとめと発展的アプローチ**

  全体の総括、運用時の注意点（チャンク分割、メタデータフィルタリング）、検索精度の評価手法（Ragas 等）についてまとめます。

1. RAGと情報検索の基本

RAGとは何か

RAG（Retrieval-Augmented Generation: 検索拡張生成）は、大規模言語モデル（LLM）に外部の知識ベースを組み合わせる技術です。LLM が回答を作る前に、質問に関連する文書やデータを外部から検索し、その見つかった情報をプロンプト（指示文）に付与して LLM に渡します。

LLM は学習した時点のデータしか知らず、最新の出来事や特定の企業内の機密文書、個人のプライベートな情報を持っていません。また、知らないことに対して、もっともらしい嘘をついてしまう現象（ハルシネーション）が起こることもあります。RAG を使うと、最新情報や独自データをその都度検索して LLM に読ませることができるため、正確で根拠のある回答を生成できるようになります。

なぜ検索の品質が重要なのか

RAG システムの回答精度は、LLM の性能だけでなく、「検索の正確さ」に大きく依存します。検索ステップで適切な文書を見つけられなければ、いくら賢い LLM であっても正しい回答を作ることはできません。

逆に、関係のない余計な文書を大量に渡してしまうと、LLM が混乱して誤った情報を回答に混ぜてしまう原因になります。また、プロンプトの文字数が増えるため、処理速度の低下や API 利用コストの増加につながります。そのため、数千・数万の文書の中から「質問の答えが書かれているピンポイントな情報」を上位に選び出す検索技術が、RAG の成功において最も重要です。

情報検索の2大アプローチ：密検索と疎検索

現在の RAG における検索技術は、大きく分けると「密検索（ベクトル検索）」と「疎検索（キーワード検索）」の2つが存在します。

密検索（Dense Retrieval）は、文章全体を意味を表現する数値の並び（密ベクトル）に変換し、質問文と文書の意味の近さ（コサイン類似度など）を計算して探す手法です。表現の揺らぎや同義語（例：「クルマ」と「自動車」）を自然に吸収できる強みがあります。

疎検索（Sparse Retrieval）は、文章に含まれる単語そのものの出現頻度や珍しさを計算して探す、昔から使われている確実な手法です。その代表格が「BM25」です。専門用語、型番、人名、固有名詞のように、完全に一致するキーワードを厳密に探したい場合に圧倒的な強みを発揮します。

近年の高精度な RAG では、この両方の長所を活かす「ハイブリッド検索」が標準的な設計となっています。次章では、疎検索の代表である BM25 のアルゴリズムと数理について詳しく見ていきます。

2. BM25アルゴリズムの仕組み

BM25とは何か

BM25（Best Matching 25）は、1990年代に情報検索の分野で開発された、伝統的かつ強力なキーワード検索スコアリングアルゴリズムです。一般的には「Okapi BM25」とも呼ばれ、オープンソースの全文検索エンジンである Elasticsearch や OpenSearch、Apache Lucene でも長年標準スコアとして採用されています。

BM25 は、古典的な検索手法である「TF-IDF（Term Frequency - Inverse Document Frequency）」を大幅に改良したものです。質問に含まれる単語が、ある文書の中にどれくらい頻繁に、そしてどれくらい特徴的に現れるかを確率統計モデルに基づいて計算します。

TF-IDFからの進化とBM25の計算式

TF-IDF では、単語の出現回数が増えれば増えるほどスコアが比例して直線的に上昇します。しかし、現実には単語が10回出てくる文書と100回出てくる文書の関連度は、10倍も違わないはずです。BM25 はこの課題を解決するために、次の数式で関連度スコアを算出します。

```
Score(D, Q) = Σ IDF(q_i) * [ (f(q_i, D) * (k1 + 1)) / (f(q_i, D) + k1 * (1 - b + b * (|D| / avgdl))) ]
```

各記号の意味は次の通りです。

- **Q**: ユーザーが入力した質問文（単語 `q_1, q_2, ...` の集まり）
- **D**: 評価対象の文書
- **f(q_i, D)**: 文書 D の中に単語 `q_i` が出現した回数（TF）
- **|D|**: 文書 D の単語数（文書長）
- **avgdl**: データベース内の全文書の平均単語数（Average Document Length）
- **k1, b**: スコアの挙動を調整するためのパラメータ

BM25を支える3つの重要な要素

BM25 が高精度に機能する理由は、次の3つの仕組みをバランスよく組み合わせているためです。

▸ 珍しい単語を高く評価する（IDF: 逆文書頻度）

「これ」「です」「あります」のような、どの文書にもよく出てくる一般的な単語は、検索の役に立ちません。一方で、「量子力学」や「LangChain」のように限られた文書にしか出てこない珍しい単語は、文書の主題を強く表しています。IDF

は、全文書中に出現する割合が低い希少な単語ほど、高い重みを与える仕組みです。

▸ 単語頻度の頭打ち効果（TFの飽和特性とパラメータ k1）

BM25 では、文書内に単語が何回も現れた場合、最初はスコアが大きく上がりますが、回数が増えるにつれてスコアの上昇が緩やかになり、やがて上限値に近づきます。これを「飽和（Saturation）」と呼びます。パラメータ `k1` は、その飽和の速さを制御する値です。通常は `1.2` から `2.0` 程度に設定され、`k1` を小さくすると早い段階でスコアが頭打ちになります。

▸ 長い文書に対するペナルティ（文書長正規化とパラメータ b）

長い文書は文字数が多いため、短い文書に比べて偶然多くの単語が含まれやすくなります。BM25 は、データベース全体の平均文書長（`avgdl`）と対象文書の長さ（`|D|`）を比較し、平均より長い文書にはスコアを割り引くペナルティを課します。パラメータ `b` はこのペナルティの強さを調整する値で、`0.0` から `1.0` の範囲で設定します。通常は `0.75` 前後が最適とされています。

日本語におけるBM25の注意点

英語は単語同士が半角スペースで区切られているため、文字列をそのまま分割して BM25 を適用できます。しかし、日本語は単語の間にスペースが入らない「わかち書き」ではない言語です。

そのため、日本語テキストで BM25 を使用する場合は、事前に「形態素解析」を行い、文章を品詞単位の単語（トークン）に正しく分割する必要があります。形態素解析ツールには、MeCab、Janome、Sudachi などがあります。適切なトークナイズを行わずに文字単位などで分割してしまうと、BM25 本来の単語頻度や珍しさが正しく計算できず、検索精度が著しく低下します。Python + LangChain で実装する際も、このトークナイザーの設定が極めて重要な鍵となります。

3. ベクトル検索とBM25の比較

密ベクトル検索（Dense Retrieval）の特徴

密ベクトル検索は、テキストを埋め込みモデル（Embedding Model: OpenAI の text-embedding-3 や BGE など）に入力し、数百から数千次元の高密度な数値配列に変換して類似度を測る手法です。

▸ 密ベクトル検索の強み

文章全体の文脈やニュアンス、同義語を理解できる点に最大の強みがあります。たとえば、「お腹が痛い」という質問に対して、「胃の不調」「腹痛の治療法」といった異なる表現が使われた文書であっても、意味が近ければ上位に検索できます。ユーザーが専門用語を知らずに自然言語で質問した際にも柔軟に対応できます。

▸ 密ベクトル検索の弱み

固有名詞、製品型番、エラーコード、人名などの完全一致を捉えることが苦手です。たとえば、「Error-404-X9」と「Error-404-X8」は、ベクトル空間上ではほぼ同一の数値配列に変換されてしまい、微妙な英数字の差異を区別できません。また、埋め込みモデルの計算コストやインデックス作成のメモリ消費量が大きい点も注意が必要です。

疎検索（Sparse Retrieval: BM25）の特徴

疎検索は、文書に含まれる具体的な単語の一致を手がかりに検索を行う手法であり、BM25 がその筆頭です。

▸ BM25の強み

入力されたキーワードそのものを厳密に検索するため、製品名、品番、プログラムの関数名、エラーコード、法令番号などの検索で抜群の正確性を誇ります。また、事前の重いベクトル計算（GPU や Embedding API の呼び出し）が不要であり、CPU のみで超高速にインデックス作成と検索処理を実行できます。

▸ BM25の弱み

単語の「意味」や「言い換え」を理解できません。「スマートフォンの電池持ち」で検索した場合、「スマホのバッテリー持続時間」と書かれた文書は単語が一致しないため、上位に表示されにくくなります。また、スペルミスや表記揺れにも弱く、ユーザーが適切なキーワードを入力できないと見当違いの結果になることがあります。

ベクトル検索とBM25の比較対照表

両者の特徴を整理すると、下表のようになります。

比較項目	密ベクトル検索（Dense）	疎検索（BM25 / Sparse）
主なマッチング方式	意味的類似度（コサイン類似度・内積）	単語の出現頻度と希少性（語彙一致）
言い換え・同義語	非常に得意（自動的に吸収）	苦手（同義語辞書等がないと不一致）

比較項目	密ベクトル検索（Dense）	疎検索（BM25 / Sparse）
固有名詞・型番・コード	苦手（わずかな文字列差異を無視しがち）	非常に得意（完全一致を確実に検出）
計算リソース	高い（Embeddingモデル計算・GPU推奨）	非常に低い（CPUのみで軽量動作）
初期構築コスト	Embedding API代やモデル準備が必要	オープンソースやローカルCPUで即時利用可能
言語依存性	多言語対応モデルなら言語横断も容易	言語ごとの形態素解析やトークナイザーが必須

なぜハイブリッド検索が必要なのか

表からも明らかなように、密ベクトル検索と BM25 は、互いの弱点を完全に補い合う関係にあります。

実際の業務データでは、「概念的な質問（例：有給休暇の申請方法について）」と「具体的な質問（例：申請様式No.12-Bの提出先）」の両方が頻繁に行われます。ベクトル検索だけに頼ると型番や固有名詞で誤検索が起き、BM25だけに頼ると表現の揺らぎや口語表現に追従できません。

そのため、両方の手法で同時に検索を行い、スコアを合成して上位の文書を抽出する「ハイブリッド検索」が、実用的なRAG におけるデファクトスタンダードとなっています。次章では、これらを実現するためのデータストアについて詳しく解説します。

4. 主要データストアの比較と選定

データストアの重要性と分類

RAG システムにおいて、文書やそのベクトルデータをどこに保存し、どう検索するかを決定するデータストアの選定は極めて重要です。検索速度、精度、インフラ運用の難易度、コストに直結するためです。

RAG で用いられるデータストアは、大きく次の4つに分類されます。

- **ローカル・組込型**: アプリケーション内やローカル環境で手軽に動く軽量ストア
- **特化型ベクトルデータベース**: ベクトル検索に最適化された高性能エンジン
- **全文検索エンジン拡張**: 強力な BM25 機能にベクトル検索機能を追加したエンジン
- **リレーショナルデータベース拡張**: 既存の RDBMS にベクトル検索拡張を追加したシステム

主要データストアの詳細比較

▸ Chroma

Chroma は、AI アプリケーション開発者向けに設計されたオープンソースの埋め込みベクトルストアです。Python から `pip install chromadb` で即座に導入でき、SQLite をバックエンドにしてローカルで動作します。

- **メリット**:

  セットアップが非常に簡単で、外部サーバーを立てずにメモリまたはローカルディスク上で動かせます。
  LangChain との統合が非常に手厚く、プロトタイプ作成や小中規模の開発を素早く開始できます。

- **デメリット**:

  大規模な分散処理や数千万件を超えるような大規模データには不向きです。標準では高度な BM25 全文検索エンジンを備えておらず、純粋なベクトルストアとして動作します。

- **おすすめのユースケース**:

  個人開発、社内 PoC（概念実証）、ローカルで完結させたい小規模 RAG アプリケーション。

▸ FAISS (Facebook AI Similarity Search)

Meta（旧 Facebook）が開発した、超高速なベクトル類似度計算のためのライブラリです。データベースというよりは計算アルゴリズムのライブラリであり、インメモリで高次元ベクトルを極限まで高速に探索します。

- **メリット**:

  世界最高峰の検索アルゴリズム（IVF、HNSW、PQ など）を搭載しており、CPU および GPU で極めて高速に動作します。数十万〜数百万のベクトルを数ミリ秒で検索できる計算性能を誇ります。

- **デメリット**:

  CRUD（作成・読み取り・更新・削除）のデータ管理機能が貧弱で、永続化やメタデータ検索を自前で実装する必要があります。BM25 のようなテキストキーワード検索機能は一切持っていません。

- **おすすめのユースケース**:

  静的なベクトルデータの一括検索、ミリ秒単位の超低レイテンシが求められる推薦システムや検索バックエンド。

▸ Qdrant

Qdrant は、Rust 言語で書かれたオープンソースの高性能ベクトル検索エンジンです。ベクトルだけでなく、メタデータ（ペイロード）に基づく高度な絞り込み検索に強みがあります。

- **メリット**:

  Rust 製のためメモリ効率が良く、非常に高速です。密ベクトルだけでなく「疎ベクトル（Sparse Vector）」にもネイティブ対応しており、Qdrant 単体でベクトル検索と BM25 相当のハイブリッド検索を完結できます。Docker で簡単にセルフホストでき、クラウド版も提供されています。

- **デメリット**:

  Elasticsearch などの成熟した全文検索エンジンに比べると、日本語の高度な形態素解析辞書の細かなカスタマイズ機能は限定的です。

- **おすすめのユースケース**:

  単一のデータベースでハイブリッド検索を実現したい本格的な本番サービス、メタデータによる厳密な権限制御が必要なシステム。

▸ Pinecone

Pinecone は、ベクトル検索に特化した完全マネージド型のクラウドサービス（DBaaS: Database as a Service）です。

- **メリット**:

  サーバーの構築、クラスタ管理、バックアップなどのインフラ運用が一切不要です。API を呼び出すだけで自動スケーリングし、数十億件規模のベクトル検索にも耐えられます。疎ベクトルと密ベクトルを組み合わせたハイブリッド検索機能も標準で備えています。

- **デメリット**:

  完全にクラウドサービスであるため、データを社外のクラウドに出せない機密性の高い要件では利用できません。データ量やクエリ数に応じた従量課金が発生するため、トラフィックが多い場合はコストが高くなる可能性があります。

- **おすすめのユースケース**:

インフラ運用保守に人員を割きたくないチーム、素早く本番環境をスケールさせたいスタートアップや Web サービス。

▸ Milvus

Milvus は、Linux Foundation 傘下で開発されているオープンソースの大規模分散ベクトルデータベースです。

- **メリット**:

  数十億件から数百億件の巨大なベクトルデータを水平分散して処理できる圧倒的なスケーラビリティを備えています。エンタープライズ向けのクラスタ運用機能が充実しています。

- **デメリット**:

  Kubernetes、etcd、MinIO、Pulsar など多くのミドルウェアを必要とするため、自前でのクラスタ構築と運用保守の難易度が非常に高いです。小規模なシステムで導入すると過剰な構成になります。

- **おすすめのユースケース**:

  大規模なエンタープライズ基盤、数千万件以上の文書を扱う巨大な企業内データレイクの RAG 基盤。

▸ Elasticsearch / OpenSearch

Elasticsearch およびそのフォークである OpenSearch は、長年世界中で使われてきた分散型全文検索エンジンのデファクトスタンダードです。

- **メリット**:

  BM25 アルゴリズムの本家であり、日本語形態素解析（kuromoji、Sudachi 等）の辞書管理やカスタマイズ機能が極めて成熟しています。近年のバージョンでは HNSW アルゴリズムによるベクトル検索（k-NN）にも対応しており、伝統的な全文検索とベクトル検索を最高品質で両立できます。

- **デメリット**:

  Java 仮想マシン（JVM）上で動作するため、大量のメモリを消費します。検索設定やクラスタ運用のパラメータが多く、学習コストと運用負荷が高めです。

- **おすすめのユースケース**:

  高度な日本語全文検索とベクトル検索を高いレベルで両立したい本番システム、既存のログ・検索基盤を活用したい環境。

▸ PostgreSQL (pgvector)

世界で最も普及しているオープンソース RDBMS の一つである PostgreSQL に、ベクトル検索機能を追加する拡張機能が `pgvector` です。

- **メリット**:

通常の業務テーブル（ユーザー情報、注文情報、権限テーブルなど）と同じデータベース内でベクトルを管理できます。SQL の `JOIN` や `WHERE` 句を使った完全なトランザクション整合性とメタデータ検索が可能です。新しいデータベースを導入・運用する必要がありません。

- **デメリット**:

  特化型ベクトル DB や FAISS と比べると、数百万件を超える大規模データでの検索速度やスループットは劣ります。PostgreSQL 標準の全文検索機能（pg_trgm や to_tsvector）は日本語の BM25 と比較すると精度面で見劣りします。

▸ SQLite FTS5 (組み込みBM25エンジン)

Python 3 に標準搭載されている SQLite の全文検索モジュール（Full-Text Search 5）です。外部サーバーの立ち上げや重い外部ライブラリのインストールが不要で、単一の `.db` ファイル上で C 言語レベルの高速な BM25 スコアリングを実行できます。

- **メリット**:

  Python 標準ライブラリのみで動作し、外部依存がゼロです。数百万件規模のテキストであっても、ディスクベースと OS ページキャッシュにより常駐メモリ数十 MB・検索レイテンシ数ミリ秒で動作します。

- **デメリット**:

  ベクトル検索（密ベクトル）機能は標準で持たないため、ベクトル検索と併用する場合は Chroma などのベクトルストアと組み合わせる必要があります。また、標準トークナイザー（unicode61）では日本語を単語分割できないため、文字バイグラム（2-gram）等の前処理を自前で実装する必要があります。

- **おすすめのユースケース**:

  外部サービスや外部ミドルウェアに依存したくないローカル／デスクトップアプリ、数百万件規模の大規模テキストに対する低リソース・高速なキーワード辞書引き（詳細は第9章の実践事例を参照）。

主要データストアの比較一覧表

各データストアの特性を一覧表にまとめます。

データストア	分類	ホスティング形態	BM25・疎検索	検索速度	運用負荷	主なメリット	主なデメリット
Chroma	組込型	ローカル / 自前	△ (外部連携)	中	最小	インストール即起動、軽量	大規模分散に不向き

データストア	分類	ホスティング形態	BM25・疎検索	検索速度	運用負荷	主なメリット	主なデメリット
FAISS	ライブラリ	インメモリ / 自前	× (非対応)	超高速	低〜中	圧倒的な計算速度、GPU対応	CRUD・永続化が自前
SQLite FTS5	組込型	ローカル / 単一DB	◯ (組み込みBM25)	高速	ゼロ	ゼロ外部依存、省メモリ	単体ではベクトル非対応
Qdrant	特化型	自前 / クラウド	◯ (疎ベクトル)	高速	中	Rust製で高速、疎密両対応	形態素辞書の調整が限定的
Pinecone	特化型	完全クラウド	◯ (疎ベクトル)	高速	ゼロ	運用不要、自動スケール	クラウド限定、従量課金
Milvus	特化型	分散クラスタ	△ (外部連携)	高速	非常に高	数十億件へのスケーラビリティ	構成要素が多く運用が複雑
Elasticsearch	全文検索	自前 / クラウド	◎ (最高峰BM25)	高速	高	成熟したBM25と日本語辞書	メモリ消費大、設定が複雑
PostgreSQL	RDBMS拡張	自前 / クラウド	△ (LIKE/trgm)	中	低 (既存活用)	既存SQL・テーブルと結合可能	大規模での性能限界

データストア選定のガイドライン

システムの要件に応じて、次の基準で選定することをおすすめします。

1. **プロトタイプ作成や手元の検証**: 手軽さを最優先し、`Chroma` または `FAISS` を採用します。
2. **インフラ運用を避けたいクラウド本番環境**: 完全マネージドな `Pinecone` または Qdrant Cloud を採用します。
3. **既存の業務システムへの RAG 組み込み**: 既存のデータベースが PostgreSQL であれば、まずは `pgvector` を導入するのが最も低コストです。
4. **単一ストアで高品質なハイブリッド検索 (BM25 + ベクトル) を実現したい場合**: 日本語辞書の細かな制御を重視するなら `Elasticsearch`、最新の特化型ベクトル検索エンジンなら `Qdrant` を選択します。

5. **外部依存なしで大規模テキストの高速 BM25 検索をローカル完結させたい場合**: Python 標準の `SQLite FTS5` を採用し、バイグラム展開と組み合わせることで、省メモリかつミリ秒応答の検索環境を実現できます（第9章・第10章を参照）。

次章からは、Python と LangChain を使って、これらを組み合わせた検索パイプラインを実際に実装していきます。

5. LangChainによるBM25の実装

LangChainにおけるBM25Retriever

LangChain には、BM25 アルゴリズムによる検索を簡単に利用できる `BM25Retriever`（`langchain_community.retrievers`）が用意されています。このコンポーネントは、内部で軽量な Python ライブラリである `rank_bm25` を使用しており、外部サーバーを立ち上げることなくローカル環境のメモリ上で高速にキーワード検索を行えます。

標準の `BM25Retriever` は、英語テキストを前提としているため、単語を空白（スペース）で区切る仕様になっています。日本語テキストをそのまま渡すと、文章全体が1つの長い単語として扱われてしまい、正しく検索できません。そのため、形態素解析ライブラリ（Janome や MeCab など）を使って「日本語トークナイズ関数」を自作し、`preprocess_func` 引数に渡すことが必須となります。

必要なパッケージのインストール

まずは、必要なライブラリをインストールします。ここでは、追加の C++ コンパイラや外部バイナリのセットアップが不要で、`pip` だけで手軽に動かせる日本語形態素解析器「Janome」を使用します。

```
pip install langchain langchain-community rank-bm25 janome
```

日本語対応BM25Retrieverの実装コード

日本語の文章群に対して BM25 でキーワード検索を行う最小限の実装コードは、次の通りです。

```python
from janome.tokenizer import Tokenizer
from langchain_community.retrievers import BM25Retriever
from langchain_core.documents import Document

# 1. Janome形態素解析器の初期化
tokenizer = Tokenizer()


def japanese_tokenizer(text: str) -> list[str]:
    """日本語テキストを形態素解析して単語リストに分割する前処理関数."""
    # 名詞、動詞、形容詞などを抽出（記号や助詞は除外することも可能）
    tokens = []
    for token in tokenizer.tokenize(text):
        pos = token.part_of_speech.split(",")[0]
        # 不要な助詞や記号を省き、自立語を中心にトークン化
        if pos in ["名詞", "動詞", "形容詞"]:
            tokens.append(token.base_form)
    return tokens


# 2. 検索対象ドキュメントの準備
documents = [
    Document(
        page_content="有給休暇の申請は、取得希望日の3日前までに勤怠管理システムから提出してください。",
        metadata={"source": "勤怠規程", "doc_id": "HR-001"},
    ),
    Document(
        page_content="深夜勤務手当は、午後10時から午前5時までの労働に対して25%増しで支給されます。",
        metadata={"source": "就業規則", "doc_id": "HR-002"},
    ),
    Document(
        page_content="社内ネットワークに接続する端末は、ウイルス対策ソフトの常時有効化が義務付けられています。",
        metadata={"source": "セキュリティガイド", "doc_id": "SEC-001"},
    ),
    Document(
        page_content="経費精算システムのエラーコード ERR-502 は、通信タイムアウトを意味します。",
        metadata={"source": "FAQ", "doc_id": "IT-101"},
    ),
]

# 3. BM25Retrieverの構築（日本語トークナイザーを指定）
```

```
bm25_retriever = BM25Retriever.from_documents(
    documents=documents,
    preprocess_func=japanese_tokenizer,
    k=2,  # 上位2件を取得
)

# 4. 検索の実行
query = "エラーコード ERR-502 の意味を知りたい"
results = bm25_retriever.invoke(query)

# 5. 検索結果の表示
print(f"検索クエリ: {query}\n")
for i, doc in enumerate(results, start=1):
    print(f"[{i}] スコア上位文書 (ID: {doc.metadata['doc_id']}):")
    print(f"    内容: {doc.page_content}")
    print(f"    出典: {doc.metadata['source']}")
```

コードのポイント解説

▸ 日本語トークナイズ関数 `japanese_tokenizer`

`preprocess_func` に渡す関数は、「テキスト文字列を受け取り、単語の文字列リストを返す」インターフェースを満たす必要があります。Janome の `token.base_form`（基本形）を取得することで、「申請した」「申請する」といった活用の違いを同じ単語として統一できます。

▸ `k` パラメータの指定

`k` は検索結果として取得する文書の件数です。デフォルトは `4` ですが、`bm25_retriever.k = 3` のように後から変更することも可能です。

▸ 固有名詞やコードに対する強さ

上記のサンプルコードでは、「ERR-502」という記号混じりのエラーコードが含まれています。ベクトル検索では見落とされがちなこのような完全一致キーワードも、BM25 であれば確実に該当の FAQ ドキュメント（`IT-101`）を最上位にヒットさせることができます。

次章では、この BM25Retriever とベクトル検索を融合させた「ハイブリッド検索」を実装します。

6. ハイブリッド検索とEnsembleRetriever

ハイブリッド検索の仕組み

ハイブリッド検索とは、密ベクトル検索（意味の近さ）と疎検索（BM25による単語一致）を同時に実行し、両方の結果を統合して最適な順位で文書を返す仕組みです。

LangChain では、複数のリトリーバーを統合するための標準コンポーネントとして `EnsembleRetriever` が提供されています。これを利用することで、ベクトル検索用のリトリーバーと BM25Retriever を簡単に1つのリトリーバーとして束ねることができます。

Reciprocal Rank Fusion（RRF）による順位統合

異なるアルゴリズムの検索スコアをそのまま単純に足し算することはできません。ベクトル検索の類似度スコア（例：0.0〜1.0）と BM25 のスコア（例：0.0〜数十）では、数値の尺度や分布が全く異なるためです。

`EnsembleRetriever` では、スコアの絶対値ではなく「検索順位（ランク）」を用いたアルゴリズムである「Reciprocal Rank Fusion（RRF: 逆順位融合法）」を採用しています。各文書の順位 r に対して、次の式で統合スコアを計算します。

```
RRF_Score(d) = Σ [ w_i / (c + r_i(d)) ]
```

ここで、w_i は各検索エンジンの重み、$r_i(d)$ はエンジン i における文書 d の順位（1位なら 1）、c は順位の影響度を滑らかにする定数（通常は 60）です。この仕組みにより、どちらかの検索で1位になった文書や、両方の検索で上位に入った文書が自然と最上位に浮上します。

ハイブリッド検索の実装コード

Chroma によるベクトル検索と BM25Retriever を組み合わせたハイブリッド検索の実装コードは、次の通りです。埋め込みモデルには Ollama でローカル実行する `bge-m3` を使用します。

事前に Ollama でモデルを取得しておきます。

```
ollama pull bge-m3
```

必要なパッケージをインストールします。

```
pip install langchain-ollama chromadb
```

```python
from janome.tokenizer import Tokenizer
from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever
from langchain_community.vectorstores import Chroma
from langchain_core.documents import Document
from langchain_ollama import OllamaEmbeddings

# 1. 形態素解析器とトークナイズ関数
tokenizer = Tokenizer()


def japanese_tokenizer(text: str) -> list[str]:
    return [t.base_form for t in tokenizer.tokenize(text)]


# 2. ドキュメント群の準備
docs = [
    Document(
        page_content="有給休暇の申請は、取得日の3日前までに勤怠システムから提出してください。",
        metadata={"id": 1, "category": "HR"},
    ),
    Document(
        page_content="夏季特別休暇は、7月から9月の間に最大3日間取得できます。",
        metadata={"id": 2, "category": "HR"},
    ),
    Document(
        page_content="経費精算のエラー ERR-502 は、通信タイムアウト時に発生します。",
        metadata={"id": 3, "category": "IT"},
    ),
    Document(
        page_content="リモートワーク時のセキュリティ対策としてVPN接続を推奨します。",
        metadata={"id": 4, "category": "SEC"},
    ),
]

# 3. BM25リトリーバーの作成（疎検索）
bm25_retriever = BM25Retriever.from_documents(
    documents=docs,
    preprocess_func=japanese_tokenizer,
    k=3,
)
```

```
# 4. Chromaベクトルストアとリトリーバーの作成（密検索）
# 埋め込みモデルにはOllamaのbge-m3を使用
embeddings = OllamaEmbeddings(model="bge-m3")
vectorstore = Chroma.from_documents(
    documents=docs,
    embedding=embeddings,
)
vector_retriever = vectorstore.as_retriever(
    search_kwargs={"k": 3},
)

# 5. EnsembleRetrieverによる統合（ハイブリッド検索）
# weightsで各リトリーバーの比率を指定（合計1.0）
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, vector_retriever],
    weights=[0.5, 0.5],
)

# 6. ハイブリッド検索の実行
query = "夏休みの取得可能日数について"
results = ensemble_retriever.invoke(query)

print(f"クエリ: {query}\n")
for i, doc in enumerate(results, start=1):
    print(f"[{i}] {doc.page_content} (カテゴリ: {doc.metadata['category']})")
```

重みパラメータ(weights)の調整方針

`EnsembleRetriever` の `weights` パラメータは、各検索手法の重要度を `[BM25の重み, ベクトル検索の重み]` のようにリストで指定します。

- キーワードの一致を重視したい場合（`weights=[0.6, 0.4]` など）

製品マニュアル、仕様書、法令文書、エラーログなど、専門用語や記号、厳密な単語一致が重視されるデータセットでは、BM25 の重みを高めに設定します。

- 意味の理解や自然文検索を重視したい場合（`weights=[0.3, 0.7]` など）

Q&A サイトのログ、カスタマーサポートの問い合わせ、自由入力された相談テキストなど、ユーザーの語彙が定まっていない場合は、ベクトル検索の重みを高くします。

- 均等配分からのスタート（`weights=[0.5, 0.5]`）

最初は `[0.5, 0.5]` の均等バランスからスタートし、実際のテストクエリ群を用いて検索精度（ヒット率など）を測定しながら微調整していくのがベストプラクティスです。

次章では、さらに検索精度を極限まで高める「リランキング」の導入と、LLM と連携した完全な RAG パイプラインを完成させます。

7. リランキングとRAGパイプラインの構築

リランキング（Re-ranking）とは

ハイブリッド検索によって、キーワードの一致と意味の類似性の両面から候補文書をバランスよく収集できるようになりました。しかし、RRF（逆順位融合）は順位のみを機械的に足し合わせるため、「質問に対して本当に直接的な答えになっているか」という深い文脈までは判別できません。

そこで、検索された上位の候補文書に対して、より高精度なニューラルモデルで再度スコアリングを行い、真に重要な文書を最上位に並び替える処理を「リランキング（Re-ranking）」と呼びます。

Bi-EncoderとCross-Encoderの違い

検索とリランキングでは、使われるモデルの構造が異なります。

‣ Bi-Encoder（通常のベクトル検索）

質問文と文書をそれぞれ独立してベクトル化します。事前に文書群をベクトル化してインデックスに保存しておけるため、数万件の文書から数ミリ秒で検索できる速度のメリットがあります。しかし、質問と文書の単語同士がどう関係しているか（相互作用）を直接比較できないため、細かな意味の整合性を捉えきれない限界があります。

‣ Cross-Encoder（リランカー）

質問文と文書を1つのペアとしてモデルに入力し、すべての単語同士の相互関係を同時に計算します。計算量は非常に多くなるため数万件の検索には使えませんが、質問に対する文書の直接的な関連度を圧倒的な高精度で判定できます。

そのため、実務では「ハイブリッド検索で候補を広めに取得（例：上位10〜20件）」し、「Cross-Encoder で上位数件（例：上位3件）に絞り込む」という「2段階検索アーキテクチャ」が最高の精度を発揮します。

完全なRAGパイプラインの実装コード

ここでは、FlashRank（軽量で高速なオープンソースのリランカー）と、LangChain の LCEL（LangChain Expression Language）を用いて、質問から検索、リランキング、回答生成までを行う完全な RAG パイプラインを実装します。ローカル実行環境として Ollama を使用し、埋め込みモデルに `bge-m3`、生成 LLM に `gemma4:12b` を利用します。

事前準備として、Ollama でモデルを取得しておきます。

```
ollama pull bge-m3
ollama pull gemma4:12b
```

必要なパッケージをインストールします。

```
pip install flashrank langchain-ollama chromadb
```

```
from janome.tokenizer import Tokenizer
from langchain.retrievers import ContextualCompressionRetriever, EnsembleRetriever
from langchain_community.document_compressors import FlashrankRerank
from langchain_community.retrievers import BM25Retriever
from langchain_community.vectorstores import Chroma
from langchain_core.documents import Document
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_ollama import ChatOllama, OllamaEmbeddings

# 1. 日本語形態素解析器の準備
tokenizer = Tokenizer()


def japanese_tokenizer(text: str) -> list[str]:
    return [t.base_form for t in tokenizer.tokenize(text)]


# 2. ナレッジベース文書群
documents = [
    Document(
        page_content="有給休暇は勤続半年後に10日付与され、申請は希望日の3日前までに社内ポータルから行います。",
        metadata={"id": "doc-01", "category": "人事"},
    ),
    Document(
        page_content="夏季休暇は7月〜9月の間に連続または分割して最大3日間取得可能です。",
        metadata={"id": "doc-02", "category": "人事"},
    ),
    Document(
        page_content="通信エラー ERR-502 はゲートウェイタイムアウトです。ルータ再起動をお試しください。",
        metadata={"id": "doc-03", "category": "情シス"},
    ),
    Document(
        page_content="社内PCにはセキュリティソフトの常時導入と月次アップデートが義務付けられています。",
        metadata={"id": "doc-04", "category": "セキュリティ"},
    ),
]

# 3. 第1段階：ハイブリッド検索の構築（広めに取得：k=4）
```

```
bm25_retriever = BM25Retriever.from_documents(
    documents=documents,
    preprocess_func=japanese_tokenizer,
    k=4,
)

embeddings = OllamaEmbeddings(model="bge-m3")
vectorstore = Chroma.from_documents(documents=documents, embedding=embeddings)
vector_retriever = vectorstore.as_retriever(search_kwargs={"k": 4})

base_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, vector_retriever],
    weights=[0.5, 0.5],
)

# 4. 第2段階：リランカー（FlashRank）の適用（上位2件に厳選）
compressor = FlashrankRerank(top_n=2)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=base_retriever,
)

# 5. プロンプトテンプレートの定義
prompt = ChatPromptTemplate.from_template(
    """あなたは親切な社内アシスタントです。以下の参考ドキュメントの内容のみに基づいて、質問にわかりやすく答えてください。
答えが見つからない場合は「提供された情報からはわかりません」と答えてください。

【参考ドキュメント】
{context}

【質問】
{question}

【回答】:"""
)



def format_docs(docs: list[Document]) -> str:
    """検索された複数の文書を1つのコンテキスト文字列に整形する関数."""
    return "\n\n".join(
        f"[{d.metadata.get('id', 'N/A')}] {d.page_content}" for d in docs
```

```
    )


# 6. LLMとパイプライン（LCEL）の結合
llm = ChatOllama(model="gemma4:12b", temperature=0)

rag_chain = (
    {"context": compression_retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

# 7. 実行
question = "夏休みは何日取れますか？"
response = rag_chain.invoke(question)

print(f"質問: {question}")
print(f"回答: {response}")
```

パイプラインの処理フローまとめ

このパイプラインを実行すると、以下の順序で安全かつ高精度に回答が生成されます。

1. **質問の入力**: ユーザーが「夏休みは何日取れますか？」と質問します。
2. **ハイブリッド検索**: BM25 と Chroma（Ollama `bge-m3` による密ベクトル）が並行して検索を実行し、RRF で上位4件の候補ドキュメントを収集します。
3. **リランキング**: Cross-Encoder（FlashRank）が質問と各候補文書を照合し、最も文脈が合致する上位2件（夏季休暇の文書など）を厳選します。
4. **プロンプト注入**: 厳選された文書がプロンプトのコンテキスト欄に注入されます。
5. **回答生成**: ローカル LLM（Ollama `gemma4:12b`）が外部知識に忠実に基づいて、ハルシネーションを起こさずに正確な回答を生成します。

次章では、実運用環境で直面する多様なデータソース（IRIS バンドルデータ、LOTUS ノートデータ、大容量テキスト行データ、複数 PDF ドキュメント）に対して、高精度な BM25 およびハイブリッド索引システムを構築・永続化する実践パイプラインを解説します。

8. 実践データソース別の索引システム構築

索引構築パイプライン（Ingestion Pipeline）の重要性

前章までで、BM25 とベクトル検索を組み合わせたハイブリッド検索やリランキングのアルゴリズムを解説しました。しかし、実際の業務システムで RAG を安定運用するためには、「元データをどのように前処理・チャンキングし、高精度な索引（インデックス）として永続化するか」というインジェスト設計が極めて重要になります。

対象となるデータソースの特性によって、求められる前処理やインデックス化のアプローチは大きく異なります。

データソース種別	主な特性と課題	推奨されるインデックス化戦略
IRIS バンドルデータ	構造化 Markdown、YAML フロントマター、コンセプト親子関係	フロントマターからメタデータを抽出、見出し構造を保持した階層チャンキング
LOTUS ノートデータ	継続的に推敲されるノート、階層ディレクトリ、頻繁な更新	更新日時やファイルハッシュによる差分検知、増分インデックス更新
大規模テキスト（行単位）	数十万〜数百万行のログや対話履歴、メモリ枯渇リスク	ジェネレータによるストリーミング読み込み、行番号保持、バッチ登録
複数 PDF ドキュメント	レイアウト構造、ページ境界、改行ノイズ、表やヘッダー	ページ単位抽出、不要な改行やヘッダーの正規化、ページ番号メタデータ付与

本章では、これら4つの実践的なデータソースを対象に、Python による具体的な索引構築パイプラインの実装手順を解説します。

必要なパッケージのセットアップ

各データソースの前処理や解析に必要なライブラリをインストールします。

```
pip install langchain langchain-community langchain-chroma langchain-ollama rank-bm25 janome pypdf pyyaml
```

共通の前処理として利用する日本語形態素解析トークナイザーを定義しておきます。

```python
from janome.tokenizer import Tokenizer

tokenizer = Tokenizer()


def japanese_tokenizer(text: str) -> list[str]:
    """日本語テキストを形態素解析して自立語を抽出するトークナイザー."""
    tokens = []
    for token in tokenizer.tokenize(text):
        pos = token.part_of_speech.split(",")[0]
        if pos in ["名詞", "動詞", "形容詞"]:
            tokens.append(token.base_form)
    return tokens
```

1. IRIS バンドルデータの索引化

特徴と課題

IRIS のナレッジバンドル（`data/bundles/<bundle-name>/wiki/`）は、YAML フロントマターでメタデータ（タグ、参照ソース、親コンセプトなど）が厳密に管理された Markdown ドキュメント群です。単にテキストとして分割するのではなく、フロントマターの属性を検索メタデータとして抽出し、見出し構造（H1, H2, H3）を意識してチャンキングすることで、親ドキュメントの文脈を失わない高精度なインデックスを構築できます。

実装手順

1. Markdown ファイルから YAML フロントマターと本文を分離して解析します。
2. `MarkdownHeaderTextSplitter` で見出し階層を保持しながら本文を分割します。
3. フロントマターのメタデータを各チャンクに引き継ぎ、BM25 およびベクトルストアへ登録します。

```python
from pathlib import Path
from typing import Any
import yaml

from langchain_community.retrievers import BM25Retriever
from langchain_core.documents import Document
from langchain_text_splitters import MarkdownHeaderTextSplitter


def parse_frontmatter_markdown(file_path: Path) -> tuple[dict[str, Any], str]:
    """YAMLフロントマターと本文を抽出する."""
    content = file_path.read_text(encoding="utf-8")
    metadata: dict[str, Any] = {}
    body = content

    if content.startswith("---"):
        parts = content.split("---", 2)
        if len(parts) >= 3:
            try:
                metadata = yaml.safe_load(parts[1]) or {}
            except Exception:
                metadata = {}
            body = parts[2].strip()

    metadata["source_file"] = str(file_path.as_posix())
    metadata["doc_id"] = file_path.stem
    return metadata, body


def index_iris_bundle(
    bundle_wiki_dir: Path,
) -> tuple[BM25Retriever, list[Document]]:
    """IRISバンドルのwikiディレクトリを走査して索引を構築する."""
    headers_to_split_on = [
        ("#", "Header_1"),
        ("##", "Header_2"),
        ("###", "Header_3"),
    ]
    markdown_splitter = MarkdownHeaderTextSplitter(
        headers_to_split_on=headers_to_split_on
    )
    all_chunks: list[Document] = []
```

```
    for md_file in bundle_wiki_dir.glob("*.md"):
        meta, body = parse_frontmatter_markdown(md_file)
        split_docs = markdown_splitter.split_text(body)

        for chunk in split_docs:
            chunk_metadata = meta.copy()
            chunk_metadata.update(chunk.metadata)
            all_chunks.append(
                Document(
                    page_content=chunk.page_content, metadata=chunk_metadata
                )
            )

    retriever = BM25Retriever.from_documents(
        documents=all_chunks,
        preprocess_func=japanese_tokenizer,
        k=3,
    )
    return retriever, all_chunks
```

2. LOTUS ノートデータの索引化と差分更新

特徴と課題

LOTUS（`data/lotus/<bundle-name>/`）は、日報やドラフト、アイデアメモなどの「生きたドキュメント」が格納される階層構造のディレクトリです。ファイルが頻繁に新規作成・推敲・更新されるため、毎回全件を再インデックスすると大きな計算コストがかかります。ファイルの最終更新日時（mtime）やハッシュ値を追跡し、**変更があったノートのみを増分更新する仕組み** が有効です。

実装手順

1. インデックス対象のファイルパスとハッシュ値を管理台帳（JSON / メモリ）に記録します。
2. 更新または新規追加されたファイルのみを読み込み、チャンキングしてインデックスに追加します。
3. 削除されたファイルのチャンクはインデックスから除外します。

```python
import hashlib
import json
from pathlib import Path
from langchain_community.retrievers import BM25Retriever
from langchain_core.documents import Document
from langchain_text_splitters import RecursiveCharacterTextSplitter


def compute_file_hash(path: Path) -> str:
    """ファイルのMD5ハッシュ値を計算する."""
    hasher = hashlib.md5()
    hasher.update(path.read_bytes())
    return hasher.hexdigest()


class LotusIncrementalIndexer:
    """LOTUSノート用の差分インデックス更新マネージャー."""

    def __init__(self, root_dir: Path, ledger_path: Path):
        self.root_dir = root_dir
        self.ledger_path = ledger_path
        self.ledger: dict[str, str] = self._load_ledger()
        self.documents: list[Document] = []
        self.splitter = RecursiveCharacterTextSplitter(
            chunk_size=400, chunk_overlap=50
        )

    def _load_ledger(self) -> dict[str, str]:
        if self.ledger_path.exists():
            return json.loads(self.ledger_path.read_text(encoding="utf-8"))
        return {}

    def _save_ledger(self) -> None:
        self.ledger_path.parent.mkdir(parents=True, exist_ok=True)
        self.ledger_path.write_text(
            json.dumps(self.ledger, indent=2, ensure_ascii=False),
            encoding="utf-8",
        )

    def scan_and_build(self) -> BM25Retriever:
        """差分を検知してBM25Retrieverを構築する."""
        current_files: dict[str, Path] = {}
```

```
        for p in self.root_dir.rglob("*.md"):
            if not p.name.startswith("."):
                current_files[str(p.as_posix())] = p

        all_docs: list[Document] = []
        new_ledger: dict[str, str] = {}

        for rel_path, file_path in current_files.items():
            file_hash = compute_file_hash(file_path)
            new_ledger[rel_path] = file_hash

            # 本文読み込みとチャンキング
            content = file_path.read_text(encoding="utf-8")
            chunks = self.splitter.split_text(content)
            for i, chunk_text in enumerate(chunks):
                all_docs.append(
                    Document(
                        page_content=chunk_text,
                        metadata={
                            "source": rel_path,
                            "chunk_id": f"{file_path.stem}_{i}",
                            "file_name": file_path.name,
                        },
                    )
                )

        self.ledger = new_ledger
        self._save_ledger()
        self.documents = all_docs

        return BM25Retriever.from_documents(
            documents=all_docs,
            preprocess_func=japanese_tokenizer,
            k=3,
        )
```

3. 大規模テキストデータ（行単位データ）のストリーミング索引化

特徴と課題

数百万行におよぶアクセスログ、チャット履歴、辞書データ、TSV/CSV などの大容量ファイルを扱う場合、ファイル全体を一度にメモリへ読み込むとメモリ枯渇（OOM: Out of Memory）を引き起こします。このようなデータには、**ジェネレータを**

用いたストリーミング処理 と、行番号やファイルオフセットをメタデータとして保持する設計が必須です。

実装手順

1. ファイルを 1 行ずつ読み込むジェネレータを作成し、メモリ使用量を最小限に抑えます。
2. 一定件数（バッチサイズ）ごとに `Document` を生成し、BM25 またはベクトルストアへ流し込みます。
3. メタデータに `line_number` を保持することで、検索ヒット時に前後の文脈をピンポイントで逆引きできるようにします。

```python
from collections.abc import Generator
from pathlib import Path
from langchain_community.retrievers import BM25Retriever
from langchain_core.documents import Document


def stream_line_documents(
    file_path: Path, batch_size: int = 1000
) -> Generator[list[Document], None, None]:
    """大容量テキストファイルをストリーミングで1行ずつ読み込みバッチ化する."""
    batch: list[Document] = []
    with file_path.open("r", encoding="utf-8") as f:
        for line_num, line in enumerate(f, start=1):
            cleaned_line = line.strip()
            if not cleaned_line:
                continue

            doc = Document(
                page_content=cleaned_line,
                metadata={
                    "source": file_path.name,
                    "line_number": line_num,
                },
            )
            batch.append(doc)

            if len(batch) >= batch_size:
                yield batch
                batch = []

    if batch:
        yield batch


def build_line_index(
    file_path: Path, max_lines: int | None = None
) -> BM25Retriever:
    """行単位テキストからBM25Retrieverを構築する."""
    all_docs: list[Document] = []
    count = 0

    for doc_batch in stream_line_documents(file_path, batch_size=5000):
```

```
        all_docs.extend(doc_batch)
        count += len(doc_batch)
        if max_lines and count >= max_lines:
            break

    return BM25Retriever.from_documents(
        documents=all_docs,
        preprocess_func=japanese_tokenizer,
        k=5,
    )
```

4. 複数 PDF ドキュメントの索引化パイプライン

特徴と課題

業務規程集や仕様書、製品マニュアルなど、複数の PDF ファイルを横断して検索したいユースケースは非常に多く存在します。しかし、PDF は以下のような特有のノイズを含んでいるため、事前のクレンジングが検索精度を大きく左右します。

- 行末の不要な改行（文の途中で改行が入る）
- ページ番号やヘッダー・フッターの混入
- 表や記号のレイアウト崩れ

実装手順

1. `pypdf`（または `PyMuPDF`）で指定ディレクトリ内の全 PDF を走査し、ページごとにテキストを抽出します。
2. 日本語の文中にある不自然な改行を結合し、テキストを正規化します。
3. `RecursiveCharacterTextSplitter` で分割し、ページ番号とファイル名をメタデータとして付与します。

```python
from pathlib import Path
import re
from langchain_community.document_loaders import PyPDFLoader
from langchain_community.retrievers import BM25Retriever
from langchain_core.documents import Document
from langchain_text_splitters import RecursiveCharacterTextSplitter


def clean_pdf_text(text: str) -> str:
    """PDF抽出テキストの改行ノイズを正規化する."""
    # 日本語文字に挟まれた単一改行を結合（文末の句読点以外）
    cleaned = re.sub(r"([^\n。！？])\n([^\n])", r"\1\2", text)
    # 連続する複数の空白を1つの空白に置換
    cleaned = re.sub(r"[ \t]+", " ", cleaned)
    return cleaned.strip()


def index_multiple_pdfs(pdf_dir: Path) -> BM25Retriever:
    """ディレクトリ配下の複数PDFファイルを走査してインデックスを構築する."""
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=500,
        chunk_overlap=80,
        separators=["\n\n", "。", "！", "？", "\n", " ", ""],
    )
    all_chunks: list[Document] = []

    for pdf_path in pdf_dir.glob("*.pdf"):
        loader = PyPDFLoader(str(pdf_path))
        pages = loader.load()

        for page in pages:
            normalized_text = clean_pdf_text(page.page_content)
            if not normalized_text:
                continue

            page_doc = Document(
                page_content=normalized_text, metadata=page.metadata
            )
            chunks = text_splitter.split_documents([page_doc])
            all_chunks.extend(chunks)

    return BM25Retriever.from_documents(
```

```
        documents=all_chunks,
        preprocess_func=japanese_tokenizer,
        k=3,
    )
```

5. インデックスの永続化と検索システムの統合

なぜインデックスの永続化が必要か

`BM25Retriever` は内部でメモリ上に転置インデックス（単語と文書の対応表）を保持しているため、Python プロセスを終了するとインデックス情報が消失します。Web サーバーの起動時や検索リクエストのたびに膨大なドキュメントを再トークナイズすると、起動や応答に多大な時間がかかってしまいます。そのため、構築した BM25 インデックスおよびベクトルストアをディスクに保存（シリアライズ）し、次回以降は瞬時に再ロードできる仕組みを導入します。

インデックスの保存とロードの実装

```python
from pathlib import Path
import pickle
from langchain_community.retrievers import BM25Retriever


def save_bm25_retriever(retriever: BM25Retriever, save_path: Path) -> None:
    """BM25Retrieverをpickleでディスクにシリアライズ保存する."""
    save_path.parent.mkdir(parents=True, exist_ok=True)
    with save_path.open("wb") as f:
        pickle.dump(retriever, f)


def load_bm25_retriever(save_path: Path) -> BM25Retriever:
    """保存されたBM25Retrieverをディスクから高速に復元する."""
    with save_path.open("rb") as f:
        retriever = pickle.load(f)
    return retriever
```

検索システム（EnsembleRetriever）への接続

永続化した BM25 インデックスと、ディスク保存された Chroma ベクトルストアを組み合わせることで、高速起動可能なハイブリッド検索システムが完成します。

```python
from pathlib import Path
from langchain.retrievers import EnsembleRetriever
from langchain_chroma import Chroma
from langchain_ollama import OllamaEmbeddings


def create_production_hybrid_retriever(
    bm25_index_path: Path,
    chroma_persist_dir: Path,
) -> EnsembleRetriever:
    """永続化されたインデックスをロードして即座にハイブリッドリトリーバーを起動する."""
    # 1. 保存済みBM25Retrieverの復元
    bm25_retriever = load_bm25_retriever(bm25_index_path)
    bm25_retriever.k = 3

    # 2. 保存済みChromaベクトルストアのロード (Ollama bge-m3)
    embeddings = OllamaEmbeddings(model="bge-m3")
    vectorstore = Chroma(
        persist_directory=str(chroma_persist_dir),
        embedding_function=embeddings,
    )
    chroma_retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

    # 3. EnsembleRetrieverで統合
    hybrid_retriever = EnsembleRetriever(
        retrievers=[bm25_retriever, chroma_retriever],
        weights=[0.5, 0.5],
    )
    return hybrid_retriever
```

このように、データソースの性質（構造化 Markdown、推敲ノート、大容量行テキスト、PDF）に応じた前処理・チャンキングを適用し、インデックスを永続化することで、実運用環境に耐えうる頑健で高精度な RAG 索引システムを構築できます。

次章では、実践ケーススタディとして、約 235 万件に及ぶ大規模英和辞書テキスト（英辞郎）を対象に、Pure Python のインメモリ限界を乗り越え、SQLite FTS5 による超高速 BM25 全文検索を実現した実プロジェクトのアーキテクチャと検証結果を解説します。

9. 実践事例：英辞郎データ（235万件）の高速BM25検索アーキテクチャ

実プロジェクトにおけるBM25の適用課題

前章までに、Python と LangChain（`rank_bm25`）を用いた BM25 検索の実装や、インデックスの永続化手法を解説しました。しかし、実際のプロダクト開発において、数百万件規模のテキストデータを扱う場合、インメモリ中心のナイーブなアプローチではメモリ枯渇や起動遅延などの重大な壁に直面します。

本章では、AI プロンプト作成支援ツール「Prompt Studio」（`apps/prompt-studio/`）において、約 160MB・2,349,580 件の英辞郎辞書テキスト（`EIJIRO-1447-UTF8.TXT`）から英語・日本語の高速な BM25 全文検索を実現するために実施した、技術選定、アーキテクチャ設計、性能検証、およびトラブルシューティングの実践記録を解説します。

1. 背景と要件

Prompt Studio では、AI プロンプト作成中に英単語や日本語訳をストレスなく補完・辞書引きできるようにするため、以下の厳しい要件が求められました。

- **対象データ**: 英辞郎 UTF-8 テキストデータ（`EIJIRO-1447-UTF8.TXT` / `EIJIRO.txt`）
- **データ規模**: ファイルサイズ約 160MB、総エントリ数 2,349,580 行（約 235 万行）
- **機能要件**:
    - 英語見出し語および日本語訳の両方からの全文検索
    - BM25 スコアリングによる関連度ソート
    - 最大 100 件の高速返却（UI 入力補完のためレスポンス数十ミリ秒以内）
    - 初回起動時に自動インデックス作成を行い、2回目以降は再インデックス不要で即時起動できること

2. アプローチの検討と決定

初期検討: Pure Python (rank_bm25) + メモリ保持の限界

当初は、第5章や第8章で紹介したような Python の軽量ライブラリ `rank_bm25`（BM25Okapi）を用いて、テキストを形態素解析・分かち書きしてインメモリで転置インデックスを保持し、`pickle` で永続化する構成を検討しました。

しかし、実データ（235万件）で検証したところ、以下の致命的な問題が判明しました。

1. **メモリフットプリントの爆発**: 235 万エントリのトークンリストおよびスコア表を Python オブジェクトとしてヒープ上に展開すると、プロセス常駐メモリが数 GB ～ 10GB 超に達し、デスクトップ環境でメモリ圧迫を引き起こす。
2. **起動時間の肥大化**: 巨大なインデックスを `pickle.load()` でデシリアライズするだけでも数十秒～数分を要し、開発時オートリロードやプロセスの起動・停止が実質不可能になる。

3. **スケーラビリティの限界**: Pure Python によるスコア計算ループでは、ヒット件数が多いクエリで数百ミリ秒〜数秒のレイテンシが発生し、キーストロークごとの即時補完に追従できない。

採用方針: SQLite FTS5 + 2-gram ハイブリッド構成

大規模なインメモリ展開を避け、OS のページキャッシュを活用したディスクベースの転置インデックスエンジンとして、Python 3 標準の **SQLite FTS5 (Full-Text Search 5)** を採用しました。

比較項目	Pure Python (`rank_bm25`)	SQLite FTS5 (採用)
外部依存	`rank-bm25` などの外部パッケージが必要	Python 3 標準組み込み（ゼロ外部依存）
常駐メモリ	数 GB 〜 10GB 超（全件ヒープ展開）	数十 MB（ディスクベース + ページキャッシュ）
起動・準備時間	起動ごとにデシリアライズで数十秒〜数分	**0 ms**（DB ファイルオープンのみ）
スコアリング	Python インタプリタ実行	SQLite 内蔵 `bm25()`（C 言語レベル実行）
永続化管理	自前の pickle キャッシュ管理が必要	単一の `.db` ファイルで完結

3. 実装アーキテクチャ

3.1 データパースと正規表現

英辞郎テキストは以下の形式で構成されています。

> ■見出し語［品詞］： 訳語

正規表現 `^■([^:]+?)(?:\s*\{([^}]+)\})?\s*:\s*(.+)$` を用いて、見出し語（`word`）、品詞（`pos`）、訳語（`translation`）をストリーミングで高精度にパースします。

3.2 日英ハイブリッド・トークナイズ（文字バイグラム + unicode61）

SQLite FTS5 標準の `tokenize = 'unicode61'` は英数字の単語境界（空白や記号）で自然にトークナイズされますが、空白区切りのない日本語（漢字・ひらがな・カタカナ）は1つの連続トークンとして扱われてしまい、部分一致検索がヒットしません。

外部の重い形態素解析エンジン（MeCab や Janome）を導入すると依存関係が肥大化するため、**文字バイグラム（2-gram）ハイブリッド方式** を自前実装しました。

1. **インデックス作成時**:
    - 日本語の文字種（ひらがな・カタカナ・漢字・全角記号等）を検出した場合、その部分のみ2文字ずつのバイグラムに展開して FTS5 カラムに格納します。
    - 例: `サイバーパンク` → `サイ イバ バー ーパ パン ンク`
    - 英語部分はそのまま格納され、FTS5 の `unicode61` が英単語単位でトークナイズします。
2. **検索クエリ時**:

- 入力クエリに日本語が含まれる場合、同様にバイグラム展開して `AND` 検索条件を構築します。
- 英語クエリの場合は、トークンを空白区切りでそのまま FTS5 クエリに渡します。

3. **FTS5 テーブル定義**:

```
CREATE VIRTUAL TABLE eijiro_fts USING fts5(
    word,
    pos UNINDEXED,
    translation,
    tokenize = 'unicode61'
);
```

3.3 BM25 スコアリングと重み付け

FTS5 組み込みの `bm25(eijiro_fts, 10.0, 1.0)` 関数を使用します。見出し語（`word`）に対するマッチの重みを `10.0`、訳語（`translation`）に対する重みを `1.0` に設定することで、見出し語が完全または部分一致した項目が検索結果の上位に優先表示されます。

```
SELECT word, pos, translation, bm25(eijiro_fts, 10.0, 1.0) AS score
FROM eijiro_fts
WHERE eijiro_fts MATCH ?
ORDER BY score
LIMIT ?;
```

> **Note**
>
> SQLite FTS5 の `bm25()` は負の値（0 に近いほどスコアが低く、負に大きいほど関連度が高い）を返します。クライアント表示時には `round(abs(score), 3)` で正のスコアに正規化して利用します。

4. 性能検証結果

実データ（2,349,580 件）を用いた実測ベンチマーク値は以下のとおりです。

評価項目	実測値	備考
インデックス構築時間	**49.4 秒**	50,000 件単位のバッチトランザクション挿入
インデックス DB サイズ	**573 MB**	元テキスト（160MB）に対し約 3.5 倍の転置インデックス
サーバー起動時間	**< 10 ms**	DB が存在すれば即時レディ状態
英語検索レスポンス	**1 ～ 3 ms**	クエリ例: `cyberpunk`（FTS5 転置インデックス直接ヒット）
日本語検索レスポンス	**20 ～ 40 ms**	クエリ例: `サイバーパンク`（バイグラム転置インデックスヒット）
プロセス常駐メモリ	**約 40 MB**	FastAPI + Uvicorn 起動時のメモリ使用量

5. 遭遇した課題と対策（トラブルシューティング）

デスクトップ Web アプリケーションへの組み込みにあたり、以下の実践的な課題に直面し、対処を行いました。

5.1 Vite 8 ウォッチャーの巨大ファイルロック問題 (EBUSY)

- **事象**: `apps/prompt-studio/data/` に 160MB のテキストデータおよび 573MB の SQLite DB を配置した際、Vite のファイル監視（chokidar / fsevents）が巨大ファイルをスキャン・排他ロックし、Windows 上で `EBUSY: resource busy or locked` エラーが発生してフロントエンドがクラッシュした。
- **対策**: `apps/prompt-studio/vite.config.ts` において、`server.watch.ignored` に `['**/data/**']` を追加し、データディレクトリを監視対象から完全に除外した。

5.2 Uvicorn のオートリロード監視ループ

- **事象**: バックエンド開発サーバー（`uvicorn main:app --reload`）が、インデックス生成中や DB アクセス時に DB ファイルの更新を検知してリロードループに陥る問題があった。
- **対策**: `apps/prompt-studio/main.py` の `uvicorn.run` にて、`reload_excludes=["data/**", "*.txt", "*.TXT", "*.db*", "*.pkl"]` を指定し、巨大ファイルやバイナリの監視を明示的に遮断した。

5.3 日本語クエリの URL エンコード

- **事象**: クライアントから日本語クエリを送信する際、シェルや HTTP クライアントの文字コード差異によってリクエストが不正となる事象が発生した。
- **対策**: クライアント側（React SPA）の `encodeURIComponent` を用いたパラメータ送信を徹底し、バックエンド側では FastAPI の `Query(..., min_length=1)` でデコード・検証を行う構成とした。

6. まとめ

235 万件規模の大規模辞書テキストであっても、Python 標準の SQLite FTS5 と文字バイグラムを活用することで、外部ミドルウェアを一切追加せずに、常駐メモリ 40MB・応答数ミリ秒の極めて軽快な BM25 検索基盤を構築できることが実証されました。

次章では、この大規模辞書データを対象に、「Chroma 等のベクトルデータベースを組み合わせたハイブリッド運用（Dense + Sparse）」の有効性とコスト、ボトルネックを検証した評価結果を解説します。

10. 実践評価：大規模データにおけるベクトル検索とBM25のハイブリッド運用設計

語彙検索と意味検索のトレードオフ

前章では、英辞郎テキスト（2,349,580 件）を対象に、SQLite FTS5 を用いて 1 ～ 3ms で応答する超高速な BM25 全文検索基盤を構築しました。この基盤は、スペルや正確な訳語の「語彙一致（完全一致・部分一致）」において極めて高いパフォーマンスを発揮します。

しかし、実際の AI アプリケーションやプロンプト作成現場では、語彙一致だけでは解決できない次のような要求が生じます。

- 「薄暗い路地裏」「幻想的な光」といった抽象的・感覚的な日本語表現から、ニュアンスに合致する英語タグ（例: `dystopian backstreet`, `ethereal lighting`）を発掘したい。
- 表記揺れや類語・同義語（例: `futuristic` と `sci-fi`）を意味的に橋渡しして検索したい。

本章では、大規模辞書データを対象に、**Chroma 等のベクトルデータベース（Dense Retrieval）を組み合わせたハイブリッド運用（BM25 + ベクトル検索）**の有効性、コスト、技術的実現性を評価・検証した記録と、導き出された「3層ハイブリッド戦略」を解説します。

1. 比較評価マトリクス（英辞郎 235万件を対象とした場合）

235 万件の全データをベクトル化して Chroma 等で管理した場合と、現行の SQLite FTS5（BM25）との詳細な比較は以下のとおりです。

評価項目	SQLite FTS5 (BM25)	Chroma + ベクトル DB (235万件)	評価・備考
初回インデックス構築時間	**約 49 秒**	**数時間 ～ 半日**	235 万回の埋め込み推論（Embedding）バッチ処理が必要
ストレージフットプリント	**573 MB**	**4 GB ～ 15 GB**	ベクトル次元数（384～1024次元）および HNSW グラフ構造による増大
プロセス常駐メモリ	**約 40 MB**	**数 GB ～ 10 GB**	HNSW ベクトルインデックスのメモリ展開が必要
検索レイテンシ	**1 ～ 3 ms**	**50 ～ 200 ms**	クエリの Embedding 計算（30～100ms）＋ ベクトル類似度計算
タイピング補完 (即時性)	**極めて良好** (即時反映)	**遅延あり** (デバウンス必須)	キー入力ごとのインクリメンタル検索には不向き
完全一致 (Exact Match)	**100% 正確**	類似概念が混ざり順位低下のリスク	辞書のスペル確認用途では語彙一致が最優先される
概念・類語検索 (Semantic)	単語・文字の一致のみ	**極めて強力**	言語化しにくいニュアンスからの検索が可能

評価項目	SQLite FTS5 (BM25)	Chroma + ベクトル DB (235万件)	評価・備考
外部依存関係	Python 3 標準組み込みのみ	`chromadb`, PyTorch / ONNX, モデル	パッケージサイズと環境依存が肥大化

2. 大規模データ全件ベクトル化におけるボトルネック分析

2.1 埋め込み計算コスト（初期インデックス構築）

235 万件のテキストを埋め込みモデル（例: `bge-m3`, `all-MiniLM-L6-v2`, `text-embedding-3-small` 等）でベクトル化する場合、ローカル CPU で 1 秒間に 50 ～ 100 件処理できたとしても、全体で **6.5 時間 ～ 13 時間** を要します。GPU を活用しても数十分～1時間は必要であり、ユーザーがツールをセットアップする際の初回起動体験として、数十秒で完了する SQLite FTS5 と比較して極めて重い負荷となります。

2.2 メモリとディスクの消費

軽量なモデル（384次元、float32）を用いた場合でも、1件あたり `384 × 4 bytes = 1.536 KB` を消費します。純粋なベクトル生データだけで `2,349,580 × 1.536 KB ≈ 3.6 GB` となります。さらに、Chroma が高速近似最近傍探索（ANN）のために構築する HNSW（Hierarchical Navigable Small World）グラフのメタデータを含めると、ディスクおよび常駐メモリの消費量は **5 GB ～ 10 GB** を超え、一般的なデスクトップツールのリソース要件として過大になります。

2.3 クエリ時のオーバーヘッドとタイピング補完の衝突

エディタの入力補完では、ユーザーが 1 文字入力するごとに数ミリ秒で候補が絞り込まれるレスポンスが理想です。ベクトル検索では、ユーザーが入力した検索語そのものを推論モデルに通してベクトル化（数十 ms）してから探索を行うため、タイピングの追従性においてレイテンシが顕著になります。

3. 推奨アーキテクチャ: 3層ハイブリッド戦略

全データを無差別にベクトル化するのではなく、データの規模と検索の目的に応じて手法を分担させる「3層ハイブリッド戦略」が最も費用対効果に優れています。

```
flowchart TD
    subgraph UI["Prompt Studio (UI / エディタ)"]
        Client["ユーザー入力 / クエリ"]
    end

    subgraph Tier1["Tier 1: 語彙辞書引き (即時補完)"]
        FTS["英辞郎 (235万件)<br/>SQLite FTS5 (BM25)"]
        FTS_Info["- 1〜3ms 即時補完<br/>- メモリ 40MB<br/>- 正確なスペル・訳語"]
    end

    subgraph Tier2["Tier 2: タグセマンティック探索"]
        ChromaStore["プロンプトタグ (数千〜数万件)<br/>Chroma (ベクトルDB)"]
        Chroma_Info["- ニュアンス・類語探索<br/>- インデックス数秒〜数分<br/>- メモリ数十MB"]
    end

    subgraph Tier3["Tier 3: 創作ブレスト・文脈生成"]
        OllamaLLM["ローカルLLM<br/>Ollama"]
        Ollama_Info["- 複雑なアイデア出し<br/>- 文脈に沿ったプロンプト生成<br/>- 追加インデックス不要"]
    end

    Client -->|タイピング即時補完| FTS
    Client -->|ニュアンス・タグ検索| ChromaStore
    Client -->|プロンプト自動構成| OllamaLLM
```

Tier 1: 大規模辞書(235万件)は SQLite FTS5 (BM25) を継続

- **用途**: 単語のスペル確認、正確な訳語検索、直接入力補完。
- **理由**: 573MB の単一ファイル、メモリ 40MB、応答 1 ～ 3ms という現行の圧倒的な利点を維持。

Tier 2: プロンプトタグ(数千～数万件)限定の Chroma 導入

- **用途**:「こういう雰囲気・衣装・背景のタグが欲しい」というセマンティック探索。
- **対象データ**: 厳選されたタグ定義群(数千～数万件程度)。
- **利点**:
    - 数万件規模であれば、ローカル CPU でも **数秒～数十秒** で埋め込み計算・インデックス構築が完了。
    - DB サイズは数十 MB、常駐メモリも 100 ～ 200MB 程度に収まる。
    - 「辞書検索」とは別に「類似タグ探索 (Semantic Tag Finder)」パネルを新設することで、目的を明確に分離可能。

Tier 3: 複雑なニュアンス・創作支援はローカルLLMに委譲

- **用途**: 「近未来のネオン街で雨に濡れるサイバーパンク少女のプロンプト構成案を作って」といった高度な文脈処理。
- **理由**: アプリケーションに Ollama などのローカル LLM が組み込まれている場合、ベクトル DB の構築・保持コストをかけずに、LLM の事前学習知識を直接活用して最適な英語タグ群をブレインストーミングさせることが可能。

4. 結論とハイブリッド運用の設計指針

本ケーススタディの評価から得られた、実運用におけるハイブリッド運用の重要指針は以下のとおりです。

1. **「何でもベクトル化」を避ける**: 数百万件規模の辞書やログデータなど、語彙の完全一致やスペル検索が主目的のデータには、BM25（特に SQLite FTS5 や Elasticsearch）がリソース・速度・精度の全方位で優れています。
2. **意味検索（ベクトル）は「データ量を絞って」局所投入する**: セマンティック検索が必要な対象（ドメイン固有のタグや高品質ナレッジの要約）に絞ってベクトル化することで、インデックス時間とリソース消費を最小限に抑えつつ、高い検索体験を提供できます。
3. **UI/UX レベルでの責務分離**: キー入力ごとのインクリメンタル補完にはミリ秒単位で動く BM25 を割り当て、ニュアンス検索にはデバウンスを効かせたベクトル検索や LLM を割り当てることで、ストレスのないユーザー体験を実現できます。

次章では、本書全体の総括とともに、チャンキングやメタデータフィルタリング、評価フレームワークを用いた継続的な検索品質の改善アプローチをまとめます。

11. まとめと発展的アプローチ

本書のまとめ

本書では、RAG システムの回答精度を劇的に向上させるための重要技術である「BM25」と「ハイブリッド検索」、主要なデータストアの選定、実運用データソースの索引化、そして大規模データにおける実践アーキテクチャについて解説しました。

- **BM25 の役割**:

  単語の出現頻度と希少性（IDF）を活かし、型番やエラーコード、固有名詞などの完全一致検索において絶対的な信頼性を発揮します。

- **密ベクトル検索の役割**:

  文章の意味的なニュアンスや同義語を理解し、自然言語による柔軟な検索を可能にします。

- **ハイブリッド検索と2段階検索**:

  BM25 とベクトル検索を融合し、さらに Cross-Encoder によるリランキングを挟むことで、速度と精度の両立を実現できます。

- **データストアの適材適所**:

  PoC では Chroma、既存資産の活用なら PostgreSQL (pgvector)、本格的なハイブリッド運用なら Qdrant や Elasticsearch など、要件に合わせた選定が重要です。

- **大規模データにおける実装・運用の現実解**:

  数百万件規模のテキストでは、Pure Python（`rank_bm25`）のインメモリ保持によるメモリ爆発や起動遅延を回避するため、SQLite FTS5 を用いた C 言語レベルの高速 BM25 と文字バイグラムの組み合わせが極めて有効です。また、「全件ベクトル化」のボトルネックを避け、語彙検索（BM25）、意味探索（局所ベクトルDB）、文脈推論（ローカルLLM）を適材適所で分担させる階層的ハイブリッド戦略が成功の鍵となります。

本番運用に向けた実践的ベストプラクティス

実際の業務環境で RAG を安定して運用するには、検索アルゴリズム以外にも以下の設計配慮が必要です。

▸ 適切なチャンキング（文書の分割設計）

長大なドキュメントをそのまま登録すると、検索の焦点がぼやけてしまいます。文字数（例：300〜500文字程度）や見出し単位で適切に分割し、前後の文脈が途切れないようオーバーラップ（50文字程度）を設定することが大切です。

▸ メタデータフィルタリングの活用

社内の機密文書やマニュアルを検索する際、ユーザーの所属部署やアクセス権限、文書の作成年度などで検索対象を事前に絞り込む「事前フィルタリング」が不可欠です。データストア選定時は、このメタデータ絞り込みが高速に機能するかを必ず確認しましょう。

▸ 検索クエリの改善（クエリ拡張やリライト）

ユーザーが入力する検索文が短すぎたり曖昧だったりする場合、LLM を使って検索キーワードを増やしたり、検索しやすい形式に言い換えたりする「クエリ拡張（Query Expansion）」を行うと検索精度がさらに向上します。

検索精度の評価手法（RAG Evaluation）

RAG の開発・改善を継続するためには、勘に頼らず数値で精度を測定する仕組みが必要です。

Ragas や TruLens といった評価フレームワークを活用すると、以下の指標を自動測定できます。

- **忠実度（Faithfulness）**: LLM の回答が、検索された参考文書の事実に基づいているか（ハルシネーションの測定）。
- **回答関連性（Answer Relevance）**: 生成された回答が、ユーザーの質問に的確に答えているか。
- **文脈適合率（Context Precision）**: 検索された文書群の中に、回答に必要な情報が過不足なく含まれているか。

ハイブリッド検索の重みパラメータ（weights）やチャンクサイズを変更した際には、これらの指標を継続的にトラッキングして最適解を導き出しましょう。