

RAG検索機能実装計画

▣ **ステータス:** 計画段階

17 **作成日:** 2025-12-17

⌚ **目的:** RAGページの検索機能と関連度計算、AIアシスタント統合の実装計画

📊 現状分析

既存実装状況

1. RAG検索ページ構造

- UIコンポーネントは完成 (`app/rag-search/`)
- フック構造は整備済み (`useRAGSearch.ts`, `useSearchFilters.ts`など)
- 検索ロジックは未実装 (TODOコメントのみ)

2. 関連度計算

- スコアリング構造は定義済み (`lib/ragSearchScoring.ts`)
- 実装は削除済み (TODOコメントのみ)

3. 埋め込みベクトル検索

- ChromaDB統合済み (`lib/entityEmbeddingsChroma.ts`, `lib/relationEmbeddingsChroma.ts`)
- 類似度検索関数は存在 (`findSimilarEntitiesChroma`, `findSimilarRelationsChroma`)
- トピック埋め込みも対応可能

4. AIアシスタント

- RAGコンテキスト取得機能あり (`getIntegratedRAGContext`)
- MCPサーバー統合なし
- オーケストレーションレイヤーなし

⌚ 実装目標

フェーズ1: RAG検索機能の実装 (関連度計算含む)

1. 検索アルゴリズムの実装

- ChromaDBを使用したベクトル類似度検索
- エンティティ、リレーション、トピックの統合検索
- フィルタリング機能の実装

2. 関連度計算の実装

- ベクトル類似度ベースの基本スコア
- メタデータマッチングによるブースト

- 新しさ (recency) による調整
- 重要度 (importance) による調整
- 重み付けスコアリング

3. 検索結果の表示

- 関連度スコアの可視化
- スコアによるソート
- 結果のグループ化 (エンティティ/リレーション/トピック)

フェーズ2: AIアシスタント統合強化

1. MCPサーバー統合

- MCPサーバーの設定と接続
- ナレッジグラフ情報の取得
- 検索結果の取得
- システム設計ドキュメントの取得

2. オーケストレーションレイヤー

- 複数情報源の統合
- コンテキストの優先順位付け
- 情報の重複排除
- 適切な情報の選択と統合

3. AIアシスタントの改善

- MCP経由での情報取得
- オーケストレーションレイヤー経由での情報統合
- より適切な回答生成



詳細実装計画

フェーズ1-1: 関連度計算アルゴリズムの実装

ファイル: [lib/ragSearchScoring.ts](#)

実装内容

1. ベクトル類似度の正規化

```
// ChromaDBから取得した距離を類似度に変換
// distance: 0.0 (完全一致) ~ 2.0 (完全不一致)
// similarity: 1.0 (完全一致) ~ 0.0 (完全不一致)
similarity = Math.max(0, 1.0 - distance)
```

2. メタデータマッチングスコア

```
// エンティティタイプ、リレーションタイプ、セマンティックカテゴリの一致  
// キーワードの一致度  
// 組織IDの一致
```

3. 新しさスコア (Recency)

```
// 更新日時が新しいほど高いスコア  
// 指数関数的減衰: score = e^(-daysSinceUpdate / decayFactor)
```

4. 重要度スコア (Importance)

```
// 検索頻度 (searchCount) によるブースト  
// トピックの重要度フラグ  
// エンティティの重要度メタデータ
```

5. 総合スコア計算

```
finalScore = (  
    similarity * weights.similarity +  
    metadataScore * weights.metadata +  
    recencyScore * weights.recency +  
    importanceScore * weights.importance  
)
```

実装タスク

- `calculateEntityScore` の実装
- `calculateRelationScore` の実装
- `calculateTopicScore` の実装
- `calculateScoreWithCustomWeights` の実装
- `adjustWeightsForQuery` の実装 (クエリタイプに応じた重み調整)

フェーズ1-2: 検索アルゴリズムの実装

ファイル: `lib/knowledgeGraphRAG.ts`

実装内容

1. クエリ埋め込み生成

```
const queryEmbedding = await generateEmbedding(queryText);
```

2. ChromaDB検索

```
// エンティティ検索
const entityResults = await findSimilarEntitiesChroma(
    queryEmbedding,
    limit,
    organizationId
);

// リレーション検索
const relationResults = await findSimilarRelationsChroma(
    queryEmbedding,
    limit,
    organizationId
);

// トピック検索（実装が必要な場合）
const topicResults = await findSimilarTopicsChroma(
    queryEmbedding,
    limit,
    organizationId
);
```

3. SQLiteから詳細情報取得

```
// ChromaDBの結果からIDを抽出
// SQLiteからエンティティ、リレーション、トピックの詳細を取得
```

4. スコアリング適用

```
// 各結果に対して関連度スコアを計算
// スコアでソート
```

5. フィルタリング適用

```
// エンティティタイプ、リレーションタイプ、組織IDでフィルタ
// 日付範囲でフィルタ
```

実装タスク

- `searchKnowledgeGraph` の実装
- `findRelatedEntities` の実装
- `findRelatedRelations` の実装

- `getKnowledgeGraphContext` の実装
- `getIntegratedRAGContext` の実装 (システム設計ドキュメント統合)
- `updateSearchFrequency` の実装

フェーズ1-3: フックの実装

ファイル: `app/rag-search/hooks/useRAGSearch.ts`

実装内容

1. 検索実行

```
const search = async (
  query: string,
  filters: SearchFilters,
  maxResults: number
) => {
  // searchKnowledgeGraph を呼び出し
  // 結果を状態に保存
  // キャッシュに保存 (オプション)
};
```

2. キャッシュ管理

```
// メモリキャッシュ
// localStorageキャッシュ
// キャッシュの有効期限管理
```

実装タスク

- `useRAGSearch` フックの検索ロジック実装
- キャッシュ機能の実装
- エラーハンドリングの改善

フェーズ2-1: MCPサーバー統合

新規ファイル: `lib/mcp/knowledgeGraphMCP.ts`

実装内容

1. MCPサーバー接続

```
// MCPサーバーへの接続設定
// 認証とセキュリティ
```

2. ナレッジグラフ情報取得

```
// エンティティ情報の取得  
// リレーション情報の取得  
// トピック情報の取得
```

3. 検索結果取得

```
// RAG検索の実行  
// 結果の取得と整形
```

実装タスク

- MCPサーバー設定ファイルの作成
- MCPクライアントの実装
- ナレッジグラフ情報取得関数の実装
- 検索結果取得関数の実装

フェーズ2-2: オーケストレーションレイヤー

新規ファイル: `lib/orchestration/ragOrchestrator.ts`

実装内容

1. 情報源の統合

```
// ナレッジグラフ情報  
// システム設計ドキュメント  
// MCP経由の情報  
// その他の情報源
```

2. コンテキストの優先順位付け

```
// 関連度スコアによる優先順位  
// 情報源の信頼度による優先順位  
// 情報の新しさによる優先順位
```

3. 情報の重複排除

```
// 同じ情報の重複を検出  
// より信頼性の高い情報を優先
```

4. 適切な情報の選択

```
// トークン数の制限  
// 関連度の閾値  
// 情報の多様性確保
```

実装タスク

- `ragOrchestrator.ts` の作成
- 情報源統合ロジックの実装
- 優先順位付けアルゴリズムの実装
- 重複排除ロジックの実装
- 情報選択ロジックの実装

フェーズ2-3: AIアシスタント統合

ファイル: `components/AIAssistantPanel.tsx`

実装内容

1. MCP経由の情報取得

```
// MCPサーバーから情報を取得  
// オーケストレーションレイヤー経由で情報を統合
```

2. コンテキストの改善

```
// より適切な情報を選択  
// コンテキストの品質向上
```

3. 回答生成の改善

```
// 統合された情報を基に回答生成  
// 出典の明記
```

実装タスク

- AIアシスタントパネルのMCP統合
- オーケストレーションレイヤーの統合
- コンテキスト生成の改善
- 回答品質の向上



技術的詳細

関連度計算の重み設定

```
export const DEFAULT_WEIGHTS: ScoringWeights = {
  similarity: 0.6,      // ベクトル類似度 (最重要)
  recency: 0.1,         // 新しさ
  importance: 0.1,      // 重要度
  metadata: 0.2,        // メタデータマッチ
};
```

検索結果の構造

```
interface KnowledgeGraphSearchResult {
  type: 'entity' | 'relation' | 'topic';
  id: string;
  score: number;           // 統合スコア (0.0-1.0)
  similarity: number;      // ベクトル類似度 (0.0-1.0)
  entity?: Entity;
  relation?: Relation;
  topic?: TopicSummary;
}
```

MCPサーバー設定

```
interface MCPServerConfig {
  name: string;
  url: string;
  apiKey?: string;
  capabilities: string[];
}
```

オーケストレーションレイヤー設定

```
interface OrchestrationConfig {
  maxTokens: number;
  minRelevanceScore: number;
  sourceWeights: {
    knowledgeGraph: number;
    designDocs: number;
    mcp: number;
  };
}
```

実装スケジュール（詳細版）

フェーズ1: RAG検索機能（2-3週間）

Week 1: 関連度計算アルゴリズムの実装

Day 1-2: 基本スコアリング関数の実装

- `normalizeSimilarity` の実装とテスト
- `calculateMetadataScore` の実装とテスト
- `calculateRecencyScore` の実装とテスト
- `calculateImportanceScore` の実装とテスト

Day 3-4: 統合スコアリング関数の実装

- `calculateEntityScore` の実装とテスト
- `calculateRelationScore` の実装とテスト
- `calculateTopicScore` の実装とテスト
- 重み付けの調整とテスト

Day 5: クエリタイプに応じた重み調整

- `adjustWeightsForQuery` の実装
- クエリタイプの検出口ジック
- 重み調整のテスト

Week 2: 検索アルゴリズムの実装

Day 1-2: 基本検索機能の実装

- `searchKnowledgeGraph` の基本実装
- ChromaDB検索の統合
- エラーハンドリングの実装

Day 3-4: N+1問題の解決

- バッチクエリ関数の実装 (`getEntitiesBatch`, `getRelationsBatch`)
- `enrichEntityResults` の実装
- `enrichRelationResults` の実装
- `enrichTopicResults` の実装

Day 5: フィルタリングとソート

- フィルタリング機能の実装
- ソート機能の実装
- パフォーマンステスト

Week 3: フックの実装と統合テスト

Day 1-2: フックの実装

- **useRAGSearch** フックの検索ロジック実装
- キャッシュ機能の実装
- エラーハンドリングの改善

Day 3-4: UI統合

- 検索結果の表示
- スコアの可視化
- フィルタリングUIの統合

Day 5: テストと最適化

- 統合テスト
- パフォーマンステスト
- バグ修正と最適化

フェーズ2: AIアシスタント統合 (2-3週間)

Week 1: MCPサーバー統合

Day 1-2: MCPサーバー設定

- MCPサーバー設定ファイルの作成
- MCPクライアントライブラリの選定と統合
- 接続テスト

Day 3-4: ナレッジグラフ情報取得

- エンティティ情報取得関数の実装
- リレーション情報取得関数の実装
- トピック情報取得関数の実装

Day 5: 検索結果取得

- RAG検索実行関数の実装
- 結果の整形と統合
- エラーハンドリングとフォールバック

Week 2: オーケストレーションレイヤーの実装

Day 1-2: 情報源統合

- 情報源の抽象化
- 情報源からの情報取得
- 情報の正規化

Day 3-4: 優先順位付けと重複排除

- 優先順位付けアルゴリズムの実装
- 重複排除ロジックの実装
- 情報選択ロジックの実装

Day 5: トークン数管理

- トークン数計算の実装
- 情報の選択と切り詰め
- テストと最適化

Week 3: AIアシスタント統合とテスト

Day 1-2: AIアシスタント統合

- MCP経由の情報取得の統合
- オーケストレーションレイヤーの統合
- コンテキスト生成の改善

Day 3-4: テストと最適化

- 統合テスト
- パフォーマンステスト
- バグ修正と最適化

Day 5: ドキュメントとリリース準備

- ドキュメントの更新
- リリースノートの作成
- 最終確認

👉 実装の優先順位

最優先 (即座に実装)

1. 基本検索機能の実装

- `searchKnowledgeGraph` の基本実装
- ChromaDB検索の統合
- エラーハンドリング

2. N+1問題の解決

- バッチクエリ関数の実装
- 結果の拡張関数の実装

3. データ整合性チェック

- ChromaDBとSQLiteのデータ同期チェック
- エラーメッセージの改善

高優先度 (1週間以内)

4. 関連度計算の実装

- 基本スコアリング関数の実装

- 統合スコアリング関数の実装

5. フィルタリング機能

- エンティティタイプ、リレーションタイプ、組織IDでのフィルタ
- 日付範囲でのフィルタ

6. キャッシュ機能

- メモリキャッシュの実装
- localStorageキャッシュの実装

中優先度（2-3週間以内）

7. UI統合

- 検索結果の表示
- スコアの可視化
- フィルタリングUI

8. パフォーマンス最適化

- 検索速度の最適化
- メモリ使用量の最適化

低優先度（将来的に検討）

9. MCPサーバー統合

- MCPサーバーの設定と接続
- ナレッジグラフ情報の取得

10. オーケストレーションレイヤー

- 情報源の統合
- 優先順位付けと重複排除

段階的実装アプローチ

アプローチ1: MVP（最小限の実装）から開始

Phase 1: 最小限の検索機能

- ベクトル類似度のみを使用した検索
- 基本的なフィルタリング
- エラーハンドリング

Phase 2: スコアリングの追加

- 関連度計算の実装
- スコアによるソート

Phase 3: 最適化

- N+1問題の解決
- キャッシュ機能の追加
- パフォーマンス最適化

Phase 4: 高度な機能

- MCPサーバー統合
- オーケストレーションレイヤー

メリット:

- 早期に動作するバージョンを提供可能
- 段階的な改善により、リスクを低減
- ユーザーフィードバックを早期に収集可能

デメリット:

- 初期バージョンの機能が限定的
- 後から機能を追加する際のリファクタリングが必要な場合がある

アプローチ2: 完全実装から開始

Phase 1: すべての機能を実装

- 関連度計算
- 検索アルゴリズム
- MCPサーバー統合
- オーケストレーションレイヤー

メリット:

- 最初から完全な機能を提供
- リファクタリングの必要が少ない

デメリット:

- 実装期間が長い
- リスクが高い
- 早期のフィードバックが得られない

推薦: アプローチ1 (MVPから開始) を推薦

☒ 代替案・代替アプローチ

代替案1: 既存の検索ライブラリの活用

内容: 既存の検索ライブラリ (例: Algolia, Elasticsearch) を活用

メリット:

- 実装が簡単
- 検証済みの技術
- メンテナンスが不要

デメリット:

- 外部サービスへの依存
- コストがかかる可能性
- カスタマイズが制限される

推奨: 現時点では推奨しない (ChromaDBが既に実装済みのため)

代替案2: SQLiteベースの簡易検索

内容: ChromaDBが利用できない場合のフォールバックとして、SQLiteベースの簡易検索を実装

メリット:

- ChromaDBが利用できない場合でも検索可能
- 実装が比較的簡単

デメリット:

- 検索精度が低い
- パフォーマンスが劣る
- 大規模データには向き

推奨: フォールバック機能として実装を検討 (低優先度)

代替案3: 外部ベクトルDBサービスの利用

内容: Pinecone、Weaviateなどの外部ベクトルDBサービスを利用

メリット:

- スケーラビリティが高い
- メンテナンスが不要
- 高可用性

デメリット:

- コストがかかる
- 外部サービスへの依存
- データのプライバシー

推奨: 将来的に大規模データが必要になった場合に検討

代替案4: ハイブリッド検索の早期実装

内容: ベクトル検索とキーワード検索を最初から統合

メリット:

- 検索精度が高い
- キーワードマッチングによる補完

デメリット:

- 実装が複雑
- パフォーマンスへの影響
- 開発期間が長い

推奨: 基本検索機能が完成してから検討

成功指標 (KPI)

フェーズ1: RAG検索機能

1. 検索精度

- 目標: 関連度スコア0.7以上の結果が80%以上
- 測定方法: テストケースによる評価

2. 検索速度

- 目標: 検索結果の表示まで1秒以内
- 測定方法: パフォーマンステスト

3. ユーザー満足度

- 目標: ユーザーフィードバックの80%以上が肯定的
- 測定方法: ユーザーフィードバックの収集

4. エラー率

- 目標: 検索エラー率1%以下
- 測定方法: エラーログの監視

フェーズ2: AIアシスタント統合

1. 回答品質

- 目標: 回答の関連度スコア0.8以上
- 測定方法: 回答品質の評価

2. 情報源の多様性

- 目標: 3つ以上の情報源から情報を取得
- 測定方法: ログの分析

3. 応答時間

- 目標: AIアシスタントの応答まで3秒以内
- 測定方法: パフォーマンステスト

△ 実装時の注意点

1. ChromaDB必須の制約

注意点:

- ChromaDBが利用できない場合、検索結果が空になる
- organizationIdが指定されていない場合、検索できない

対策:

- ChromaDB接続確認を実装
- エラーメッセージを明確に表示
- フォールバック機能の検討

2. データ整合性

注意点:

- ChromaDBとSQLiteのデータが同期されていない可能性
- 埋め込みベクトルが更新されていない可能性

対策:

- データ同期チェック機能の実装
- 埋め込みベクトルの再生成機能
- エラーログの記録と監視

3. N+1問題

注意点:

- 検索結果ごとに個別にDBクエリを実行すると、パフォーマンスが低下

対策:

- バッチクエリの実装
- データの事前取得
- キャッシュの活用

4. パフォーマンス

注意点:

- 大量データでの検索速度の低下
- メモリ使用量の増加

対策:

- 検索結果数の制限
- ページネーションの実装
- インデックスの最適化

- キャッシュの活用

5. スコアリングの精度

注意点:

- 重み付けの調整が難しく、不適切な結果が表示される可能性

対策:

- A/Bテストによる重み付けの最適化
- ユーザーフィードバックの収集
- 機械学習による重み付けの自動調整（将来的に）

🎓 学習リソース

技術ドキュメント

1. ChromaDB

- [ChromaDB公式ドキュメント](#)
- [ChromaDB Python API](#)

2. ベクトル検索

- [近似最近傍検索（ANN）](#)
- [HNSWアルゴリズム](#)

3. RAG (Retrieval-Augmented Generation)

- [RAG論文](#)
- [RAG実装ガイド](#)

4. MCP (Model Context Protocol)

- [MCP仕様](#)
- [MCP実装ガイド](#)

参考実装

1. 既存の検索実装

- [lib/entityEmbeddingsChroma.ts](#) - エンティティ検索の実装例
- [lib/relationEmbeddingsChroma.ts](#) - リレーション検索の実装例

2. スコアリング実装

- [lib/ragSearchScoring.ts](#) - スコアリング構造の定義
- [lib/pageEmbeddings.ts](#) - ページ検索のスコアリング例

3. AIアシスタント実装

- [components/AIAssistantPanel.tsx](#) - AIアシスタントの実装例

- `lib/knowledgeGraphRAG.ts` - RAGコンテキスト取得の実装例



まとめ

実装のポイント

1. **段階的な実装**: MVPから開始し、段階的に機能を追加
2. **パフォーマンス重視**: N+1問題の回避、バッチクエリの活用
3. **エラーハンドリング**: ChromaDB必須の制約を考慮したエラーハンドリング
4. **データ整合性**: ChromaDBとSQLiteのデータ同期を確保
5. **ユーザーエクスペリエンス**: 検索結果の可視化、フィルタリング機能

推奨実装順序

1. **Phase 1**: 基本検索機能（ベクトル類似度のみ）
2. **Phase 2**: スコアリングの追加（関連度計算）
3. **Phase 3**: 最適化（N+1問題の解決、キャッシュ）
4. **Phase 4**: 高度な機能（MCP統合、オーケストレーション）

リスク軽減策

1. **ChromaDB必須の制約**: 接続確認、エラーメッセージの改善
2. **データ整合性**: 同期チェック、再生成機能
3. **N+1問題**: バッチクエリ、事前取得
4. **パフォーマンス**: 結果数制限、ページネーション、キャッシュ

成功の鍵

1. **段階的な実装**: 早期に動作するバージョンを提供
2. **ユーザーフィードバック**: 早期のフィードバック収集と改善
3. **パフォーマンス監視**: 繙続的なパフォーマンス監視と最適化
4. **ドキュメント**: 実装の詳細なドキュメント化



テスト計画

ユニットテスト

- 関連度計算関数のテスト
- 検索アルゴリズムのテスト
- フィルタリング機能のテスト

統合テスト

- RAG検索ページのE2Eテスト
- AIアシスタント統合のテスト
- MCPサーバー統合のテスト

パフォーマンステスト

- 検索速度の測定
- キャッシュ効果の測定
- 大量データでの検索性能

参考資料

- ChromaDB統合ドキュメント
- 埋め込み保存場所
- RAG検索リファクタリング計画

メリット・デメリット分析

メリット

フェーズ1: RAG検索機能

1. 検索精度の向上

- ベクトル類似度による意味的検索が可能
- メタデータマッチングによる精度向上
- 新しさ・重要度を考慮した結果ランキング

2. ユーザー体験の向上

- 関連度スコアの可視化により、ユーザーが結果の信頼性を判断可能
- フィルタリング機能により、目的の情報に素早くアクセス
- キャッシュにより、繰り返し検索が高速化

3. パフォーマンス

- ChromaDBのHNSWアルゴリズムにより、大規模データでも高速検索
- キャッシュにより、同じクエリの再検索が高速化
- 並列検索により、複数タイプの検索を同時実行

4. 拡張性

- 重み付けスコアリングにより、将来的な調整が容易
- フィルタリング機能により、新しい検索条件の追加が容易

フェーズ2: AIアシスタント統合

1. 回答品質の向上

- MCPサーバー経由で、より多様な情報源から情報を取得
- オーケストレーションレイヤーにより、最適な情報を選択
- 重複排除により、冗長な情報を削減

2. 情報源の統合

- ナレッジグラフ、システム設計ドキュメント、MCP経由の情報を統合

- 複数の情報源から最適な情報を選択

3. 保守性の向上

- オーケストレーションレイヤーにより、情報源の追加・変更が容易
- MCPサーバーにより、外部システムとの統合が容易

✖ デメリット・課題

フェーズ1: RAG検索機能

1. 実装の複雑さ

- 複数のスコアリング要素を統合する必要がある
- 重み付けの調整が難しい（試行錯誤が必要）
- フィルタリングとスコアリングの統合が複雑

2. パフォーマンス懸念

- 複数の情報源からデータを取得する必要がある（ChromaDB + SQLite）
- N+1問題の可能性（各結果に対して個別にDBクエリ）
- 大量データでのスコアリング計算が重い可能性

3. データ整合性

- ChromaDBとSQLiteのデータ同期が必要
- 埋め込みベクトルの更新が必要な場合、再生成が必要
- organizationIdが指定されていない場合、検索できない

4. キャッシュ管理

- キャッシュの無効化タイミングの判断が難しい
- メモリ使用量の増加
- キャッシュの有効期限管理が複雑

フェーズ2: AIアシスタント統合

1. MCPサーバーの依存

- MCPサーバーが利用できない場合のフォールバックが必要
- ネットワーク遅延の影響
- 認証・セキュリティの実装が必要

2. オーケストレーションレイヤーの複雑さ

- 複数の情報源を統合するロジックが複雑
- 優先順位付けのアルゴリズムが複雑
- 重複排除のアルゴリズムが複雑

3. トークン数の制限

- LLMのトークン数制限により、情報の選択が必要
- 情報の重要度判断が難しい
- 情報の多様性と関連性のバランスが難しい

⚠️ リスク分析

🔴 高リスク

1. ChromaDB必須の制約

リスク: ChromaDBが利用できない場合、検索結果が空になる

影響:

- ユーザーが検索できない
- アプリケーションの主要機能が使用不能

対策:

- ChromaDB Serverの起動確認を実装
- エラーメッセージを明確に表示
- ChromaDB起動の自動化を検討
- フォールバック機能の実装を検討 (SQLiteベースの簡易検索)

実装例:

```
// ChromaDB接続確認
async function ensureChromaDBConnection(): Promise<boolean> {
  try {
    await callTauriCommand('chromadb_health_check');
    return true;
  } catch (error) {
    console.error('ChromaDB接続エラー:', error);
    // ユーザーに通知
    showErrorNotification('ChromaDBに接続できません。検索機能が使用できません。');
    return false;
  }
}
```

2. データ整合性の問題

リスク: ChromaDBとSQLiteのデータが同期されていない

影響:

- 検索結果が0件になる
- 古いデータが表示される
- データの不整合

対策:

- データ同期チェック機能の実装
- 埋め込みベクトルの再生成機能
- データ整合性チェックの定期実行
- エラーログの記録と監視

実装例:

```
// データ整合性チェック
async function checkDataConsistency(organizationId: string): Promise<{
  entities: { chroma: number; sqlite: number };
  relations: { chroma: number; sqlite: number };
  inconsistent: string[];
}> {
  // ChromaDBとSQLiteのデータ数を比較
  // 不一致を検出
  // 不一致の詳細を返す
}
```

3. N+1問題

リスク: 検索結果ごとに個別にDBクエリを実行

影響:

- 検索速度の低下
- データベース負荷の増加
- ユーザーエクスペリエンスの悪化

対策:

- バッチクエリの実装 (IN句を使用)
- データの事前取得 (JOINを使用)
- キャッシュの活用

実装例:

```
// バッチクエリの実装
async function getEntitiesBatch(entityIds: string[]): Promise<Entity[]>
{
  // 単一のクエリで複数のエンティティを取得
  const query = `
    SELECT * FROM entities
    WHERE id IN (${entityIds.map(() => '?').join(',')})
  `;
  return await db.all(query, entityIds);
}
```



中リスク

4. パフォーマンス問題

リスク: 大量データでの検索速度の低下

影響:

- ユーザー体験の悪化
- タイムアウトエラーの発生

対策:

- 検索結果数の制限
- ページネーションの実装
- インデックスの最適化
- キャッシュの活用

実装例:

```
// 検索結果数の制限
const MAX_SEARCH_RESULTS = 100;
const results = await searchKnowledgeGraph(query, filters,
MAX_SEARCH_RESULTS);
```

5. スコアリングの精度

リスク: 重み付けの調整が難しく、不適切な結果が表示される

影響:

- 検索精度の低下
- ユーザーの不満

対策:

- A/Bテストによる重み付けの最適化
- ユーザーフィードバックの収集
- 機械学習による重み付けの自動調整（将来的に）

実装例:

```
// 重み付けの動的調整
function adjustWeightsForQuery(queryText: string): ScoringWeights {
    // クエリタイプに応じて重みを調整
    if (queryText.includes('最新')) {
        return { ...DEFAULT_WEIGHTS, recency: 0.3 };
    }
}
```

```
if (queryText.includes('重要')) {
    return { ...DEFAULT_WEIGHTS, importance: 0.3 };
}
return DEFAULT_WEIGHTS;
```

6. MCPサーバーの可用性

リスク: MCPサーバーが利用できない場合のフォールバック

影響:

- AIアシスタントの機能低下
- 情報取得の失敗

対策:

- フォールバック機能の実装（既存のRAG検索を使用）
- タイムアウトの設定
- リトライ機能の実装

実装例:

```
// MCPサーバーへの接続（フォールバック付き）
async function getMCPInfo(query: string): Promise<string> {
    try {
        const result = await mcpClient.query(query, { timeout: 5000 });
        return result;
    } catch (error) {
        console.warn('MCPサーバーへの接続に失敗。フォールバックを使用。', error);
        // 既存のRAG検索を使用
        return await getKnowledgeGraphContext(query);
    }
}
```

● 低リスク

7. キャッシュの無効化

リスク: キャッシュが古いデータを返す

影響:

- 古い検索結果が表示される
- データの不整合

対策:

- キャッシュの有効期限の設定

- データ更新時のキャッシュ無効化
- キャッシュキーの適切な設計

実装例:

```
// キャッシュの有効期限管理
const CACHE_TTL = 5 * 60 * 1000; // 5分
const cacheKey = `search:${query}: ${JSON.stringify(filters)}`;
const cached = cache.get(cacheKey);
if (cached && Date.now() - cached.timestamp < CACHE_TTL) {
  return cached.results;
}
```

🔧 より良くするための改善案

改善案1: ハイブリッド検索の実装

目的: ベクトル検索とキーワード検索を組み合わせて精度向上

実装内容:

```
// ハイブリッド検索の実装
async function hybridSearch(
  queryText: string,
  limit: number = 10
): Promise<KnowledgeGraphSearchResult[]> {
  // 1. ベクトル検索（意味的検索）
  const vectorResults = await searchKnowledgeGraph(queryText, {}, limit * 2);

  // 2. キーワード検索（BM25アルゴリズム）
  const keywordResults = await keywordSearch(queryText, limit * 2);

  // 3. 結果の統合と重複排除
  const combined = combineResults(vectorResults, keywordResults);

  // 4. スコアの再計算（ハイブリッドスコア）
  const scored = combined.map(result => ({
    ...result,
    score: calculateHybridScore(result, vectorResults, keywordResults)
  }));

  // 5. ソートと上位N件を返す
  return scored.sort((a, b) => b.score - a.score).slice(0, limit);
}
```

メリット:

- 検索精度の向上
- キーワードマッチングによる補完

デメリット:

- 実装の複雑さ
- パフォーマンスへの影響 (2回の検索が必要)

優先度: 中 (将来的に検討)

改善案2: 学習機能の実装

目的: ユーザーフィードバックから学習して検索精度を向上

実装内容:

```
// ユーザーフィードバックの記録
interface SearchFeedback {
  query: string;
  resultId: string;
  feedback: 'positive' | 'negative';
  timestamp: Date;
}

// フィードバックに基づく重み付けの調整
function adjustWeightsFromFeedback(
  feedbacks: SearchFeedback[]
): ScoringWeights {
  // フィードバックを分析
  // 重み付けを調整
  // 調整された重みを返す
}
```

メリット:

- 検索精度の継続的向上
- ユーザー体験の向上

デメリット:

- 実装の複雑さ
- データの蓄積が必要

優先度: 低 (将来的に検討)

改善案3: リアルタイム検索結果の更新

目的: WebSocket経由でリアルタイムに検索結果を更新

実装内容:

```

// WebSocket経由のリアルタイム検索
function useRealtimeSearch(query: string) {
  const [results, setResults] = useState<KnowledgeGraphSearchResult[]>([]);
  useEffect(() => {
    const ws = new WebSocket('ws://localhost:8080/search');
    ws.onmessage = (event) => {
      const newResults = JSON.parse(event.data);
      setResults(prev => [...prev, ...newResults]);
    };
    ws.send(JSON.stringify({ query }));
  }, [query]);
  return results;
}

```

メリット:

- ユーザー体験の向上
- 検索結果の即時表示

デメリット:

- WebSocketサーバーの実装が必要
- 複雑さの増加

優先度: 低 (将来的に検討)

改善案4: 検索結果の説明生成

目的: 検索結果に対して、なぜその結果が表示されたかの説明を生成

実装内容:

```

// 検索結果の説明生成
interface SearchResultExplanation {
  resultId: string;
  reasons: string[];
  scoreBreakdown: {
    similarity: number;
    metadata: number;
    recency: number;
    importance: number;
  };
}

```

```
function generateExplanation(  
    result: KnowledgeGraphSearchResult  
) : SearchResultExplanation {  
    // スコアの内訳を分析  
    // 理由を生成  
    // 説明を返す  
}
```

メリット:

- ユーザーの理解向上
- 検索結果の信頼性向上

デメリット:

- 実装の複雑さ
- パフォーマンスへの影響

優先度: 中 (将来的に検討)

改善案5: 検索履歴の分析と改善

目的: 検索履歴を分析して、検索パターンを理解し改善

実装内容:

```
// 検索履歴の分析  
interface SearchAnalytics {  
    popularQueries: string[];  
    zeroResultQueries: string[];  
    averageResultsPerQuery: number;  
    commonFilters: Record<string, number>;  
}  
  
function analyzeSearchHistory(  
    history: SearchHistory[]  
) : SearchAnalytics {  
    // 検索履歴を分析  
    // 統計情報を返す  
}
```

メリット:

- 検索機能の改善
- ユーザー行動の理解

デメリット:

- データの蓄積が必要
- プライバシーの考慮が必要

優先度: 低 (将来的に検討)

▣ 具体的な実装手順 (詳細版)

フェーズ1-1: 関連度計算アルゴリズムの実装 (詳細)

ステップ1: ベクトル類似度の正規化関数の実装

```
// lib/ragSearchScoring.ts

/**
 * ChromaDBの距離を類似度に変換
 * @param distance ChromaDBから取得した距離 (0.0-2.0)
 * @returns 類似度 (0.0-1.0)
 */
export function normalizeSimilarity(distance: number): number {
    // ChromaDBの距離は0.0 (完全一致) ~2.0 (完全不一致)
    // 類似度は1.0 (完全一致) ~0.0 (完全不一致)
    return Math.max(0, Math.min(1, 1.0 - distance / 2.0));
}
```

ステップ2: メタデータマッチングスコアの実装

```
/**
 * メタデータマッチングスコアを計算
 */
export function calculateMetadataScore(
    queryText: string,
    entity: Entity,
    filters?: SearchFilters
): number {
    let score = 0;

    // エンティティタイプの一致
    if (filters?.entityType && entity.type === filters.entityType) {
        score += 0.3;
    }

    // 組織IDの一致
    if (filters?.organizationId && entity.organizationId ===
filters.organizationId) {
        score += 0.2;
    }

    // キーワードの一致 (エンティティ名、エイリアス)
    const queryLower = queryText.toLowerCase();
    if (entity.name.toLowerCase().includes(queryLower)) {
        score += 0.3;
    }
}
```

```

    if (entity.aliases?.some(alias =>
      alias.toLowerCase().includes(queryLower))) {
        score += 0.2;
    }

    return Math.min(1.0, score);
}

```

ステップ3: 新しさスコアの実装

```

/**
 * 新しさスコアを計算 (指數関数的減衰)
 */
export function calculateRecencyScore(
  updatedAt: string,
  decayFactor: number = 30 // 30日で半減
): number {
  const now = new Date();
  const updated = new Date(updatedAt);
  const daysSinceUpdate = (now.getTime() - updated.getTime()) / (1000 *
  60 * 60 * 24);

  // 指數関数的減衰: e^(-daysSinceUpdate / decayFactor)
  return Math.exp(-daysSinceUpdate / decayFactor);
}

```

ステップ4: 重要度スコアの実装

```

/**
 * 重要度スコアを計算
 */
export function calculateImportanceScore(
  entity: Entity,
  searchCount?: number
): number {
  let score = 0.5; // ベーススコア

  // 検索頻度によるブースト
  if (searchCount !== undefined) {
    // 対数スケールで正規化 (0-1)
    score += Math.min(0.3, Math.log10(searchCount + 1) / 10);
  }

  // メタデータの重要度フラグ
  if (entity.metadata?.importance === 'high') {
    score += 0.2;
  }
}

```

```
    return Math.min(1.0, score);
}
```

ステップ5: 統合スコア計算の実装

```
/**  
 * エンティティの統合スコアを計算  
 */  
export function calculateEntityScore(  
  similarity: number,  
  entity: Entity,  
  weights: ScoringWeights = DEFAULT_WEIGHTS,  
  filters?: SearchFilters,  
  searchCount?: number  
): number {  
  // 各スコアを計算  
  const metadataScore = calculateMetadataScore('', entity, filters);  
  const recencyScore = calculateRecencyScore(entity.updatedAt ||  
entity.createdAt);  
  const importanceScore = calculateImportanceScore(entity, searchCount);  
  
  // 重み付けして統合  
  const finalScore = (  
    similarity * weights.similarity +  
    metadataScore * weights.metadata +  
    recencyScore * weights.recency +  
    importanceScore * weights.importance  
  );  
  
  return Math.min(1.0, finalScore);  
}
```

フェーズ1-2: 検索アルゴリズムの実装（詳細）

ステップ1: クエリ埋め込み生成とChromaDB検索

```
// lib/knowledgeGraphRAG.ts  
  
export async function searchKnowledgeGraph(  
  queryText: string,  
  limit: number = 10,  
  filters?: SearchFilters,  
  useCache: boolean = true,  
  timeoutMs: number = 10000  
): Promise<KnowledgeGraphSearchResult[]> {  
  // 1. クエリ埋め込み生成  
  const { generateEmbedding } = await import('./embeddings');  
  const queryEmbedding = await generateEmbedding(queryText);
```

```

// 2. 並列で各タイプを検索
const [entityResults, relationResults, topicResults] = await
Promise.all([
    // エンティティ検索
    findSimilarEntitiesChroma(queryText, limit, filters?.organizationId)
        .then(results => enrichEntityResults(results, filters)),

    // リレーション検索
    findSimilarRelationsChroma(queryText, limit,
filters?.organizationId)
        .then(results => enrichRelationResults(results, filters)),

    // トピック検索（実装が必要な場合）
    findSimilarTopicsChroma(queryText, limit, filters?.organizationId)
        .then(results => enrichTopicResults(results, filters))
]);

// 3. 結果を統合
const allResults = [
    ...entityResults,
    ...relationResults,
    ...topicResults
];

// 4. スコアでソート
allResults.sort((a, b) => b.score - a.score);

// 5. 上位N件を返す
return allResults.slice(0, limit);
}

```

ステップ2: エンティティ結果の拡張 (N+1問題の回避)

```

/**
 * エンティティ結果を拡張（バッチクエリでN+1問題を回避）
 */
async function enrichEntityResults(
    chromaResults: Array<{ entityId: string; similarity: number }>,
    filters?: SearchFilters
): Promise<KnowledgeGraphSearchResult[]> {
    if (chromaResults.length === 0) return [];

    // バッチでエンティティを取得（N+1問題を回避）
    const entityIds = chromaResults.map(r => r.entityId);
    const entities = await getEntitiesBatch(entityIds);

    // エンティティIDでマップを作成
    const entityMap = new Map(entities.map(e => [e.id, e]));

    // 検索頻度を取得（バッチ）

```

```

const searchCounts = await getSearchCountsBatch(entityIds);
const searchCountMap = new Map(searchCounts.map(s => [s.id, s.count]));

// 結果を構築
return chromaResults
  .map(({ entityId, similarity }) => {
    const entity = entityMap.get(entityId);
    if (!entity) return null;

    // フィルタリング
    if (filters?.entityType && entity.type !== filters.entityType) {
      return null;
    }
    if (filters?.organizationId && entity.organizationId !== filters.organizationId) {
      return null;
    }

    // スコア計算
    const normalizedSimilarity = normalizeSimilarity(1.0 - similarity);
    const score = calculateEntityScore(
      normalizedSimilarity,
      entity,
      DEFAULT_WEIGHTS,
      filters,
      searchCountMap.get(entityId)
    );

    return {
      type: 'entity' as const,
      id: entityId,
      score,
      similarity: normalizedSimilarity,
      entity,
    };
  })
  .filter((r): r is KnowledgeGraphSearchResult => r !== null);
}

```

ステップ3: バッチクエリ関数の実装

```

/**
 * 複数のエンティティをバッチで取得 (N+1問題を回避)
 */
async function getEntitiesBatch(entityIds: string[]): Promise<Entity[]>
{
  if (entityIds.length === 0) return [];

  // Tauriコマンド経由でSQLiteから取得

```

```
const { callTauriCommand } = await import('./localFirebase');
const results = await callTauriCommand('get_entities_batch', {
  entityIds });
return results;
}
```

⟳ 今後の拡張

1. ハイブリッド検索

- ベクトル検索 + キーワード検索
- BM25アルゴリズムの統合

2. 学習機能

- ユーザーフィードバックからの学習
- 検索結果の改善

3. マルチモーダル検索

- 画像検索
- 音声検索

4. リアルタイム更新

- WebSocket経由のリアルタイム検索結果更新
- ストリーミング検索結果