データベース構造最適化提案

現状の問題点

1. SQLiteテーブル構造の問題

topicsテーブル

- **重複ID**: `id`と`topicId`が同じ値（冗長）
- **JSON文字列**: `metadata`, `aliases`, `keywords`, `tags`がTEXT型でJSON文字列として保存（検索が非効率）
- **コンテンツ検索**: `content`が長文で、全文検索が非効率
- **RAG検索での不足**: 検索結果に`title`と`content`が含まれていない

entitiesテーブル

- **JSON文字列**: `metadata`, `aliases`がTEXT型でJSON文字列として保存（検索が非効率）
- **関連情報不足**: リレーションやトピックとの関連情報が取得しにくい

relationsテーブル

- **JSON文字列**: `metadata`がTEXT型でJSON文字列として保存（検索が非効率）
- **関連情報不足**: エンティティ名やトピック情報が取得しにくい

2. ChromaDBコレクション構造の問題

コレクション名

- **命名規則違反**: `organizationId`が空文字列の場合、`topics_`になる（末尾が`_`）
- **組織横断検索**: 空文字列の場合の処理が不適切

メタデータ

- **重複情報**: SQLiteのIDとChromaDBのIDが同じ（冗長）
- **検索に不要な情報**: `embeddingModel`, `embeddingVersion`などがメタデータに含まれている

3. RAG検索の問題

検索結果の不足

- **トピック情報**: `topicId`と`meetingNoteId`のみで、`title`や`content`が含まれていない
- **エンティティ情報**: 関連するリレーションやトピック情報が不足
- **リレーション情報**: 関連するエンティティ名が不足

コンテキスト生成

- **不要な情報**: すべてのメタデータが含まれている（トークン消費が無駄）

- **構造化不足**: 情報が構造化されていない（LLMが理解しにくい）

4. AIアシスタントへのリクエストの問題

コンテキスト長

- **長すぎる**: すべての情報が含まれている（トークン制限に達しやすい）
- **優先度不明**: 重要な情報と不要な情報が区別されていない

最適化提案

1. SQLiteテーブル構造の最適化

topicsテーブル

```
CREATE TABLE IF NOT EXISTS topics (
    id TEXT PRIMARY KEY,  -- topicIdと同じ値（後方互換性のため保持）
    topicId TEXT NOT NULL UNIQUE,  -- 主キーとして使用
    meetingNoteId TEXT NOT NULL,
    organizationId TEXT,
    companyId TEXT,
    title TEXT NOT NULL,
    description TEXT,
    content TEXT,
    semanticCategory TEXT,
    keywords TEXT,  -- JSON配列（検索用にインデックス追加を検討）
    tags TEXT,  -- JSON配列（検索用にインデックス追加を検討）

    -- RAG検索用の追加カラム
    contentSummary TEXT,  -- contentの要約（200文字程度）
    searchableText TEXT,  -- title + description + contentSummaryを結合し
た検索用テキスト

    -- ChromaDB同期状態
    chromaSynced INTEGER DEFAULT 0,
    chromaSyncError TEXT,
    lastChromaSyncAttempt TEXT,

    -- タイムスタンプ
    createdAt TEXT NOT NULL,
    updatedAt TEXT NOT NULL,

    FOREIGN KEY (meetingNoteId) REFERENCES meetingNotes(id),
    FOREIGN KEY (organizationId) REFERENCES organizations(id),
    FOREIGN KEY (companyId) REFERENCES companies(id),
    CHECK ((organizationId IS NOT NULL AND companyId IS NULL) OR
           (organizationId IS NULL AND companyId IS NOT NULL))
);

-- 検索用インデックス
CREATE INDEX IF NOT EXISTS idx_topics_searchable_text ON
```

```
topics(searchableText);
CREATE INDEX IF NOT EXISTS idx_topics_organizationId ON
topics(organizationId);
CREATE INDEX IF NOT EXISTS idx_topics_meetingNoteId ON
topics(meetingNoteId);
CREATE INDEX IF NOT EXISTS idx_topics_semanticCategory ON
topics(semanticCategory);
```

entitiesテーブル

```
CREATE TABLE IF NOT EXISTS entities (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    type TEXT NOT NULL,
    aliases TEXT,  -- JSON配列（検索用にインデックス追加を検討）
    metadata TEXT,  -- JSONオブジェクト（検索用にインデックス追加を検討）
    organizationId TEXT,
    companyId TEXT,

    -- RAG検索用の追加カラム
    searchableText TEXT,  -- name + aliases + metadataの重要フィールドを結
合した検索用テキスト
    displayName TEXT,  -- 表示用の名前（name + 重要なメタデータ）

    -- ChromaDB同期状態
    chromaSynced INTEGER DEFAULT 0,
    chromaSyncError TEXT,
    lastChromaSyncAttempt TEXT,

    -- タイムスタンプ
    createdAt TEXT NOT NULL,
    updatedAt TEXT NOT NULL,

    FOREIGN KEY (organizationId) REFERENCES organizations(id),
    FOREIGN KEY (companyId) REFERENCES companies(id),
    CHECK ((organizationId IS NOT NULL AND companyId IS NULL) OR
           (organizationId IS NULL AND companyId IS NOT NULL))
);

-- 検索用インデックス
CREATE INDEX IF NOT EXISTS idx_entities_searchable_text ON
entities(searchableText);
CREATE INDEX IF NOT EXISTS idx_entities_organizationId ON
entities(organizationId);
CREATE INDEX IF NOT EXISTS idx_entities_type ON entities(type);
```

relationsテーブル

```
CREATE TABLE IF NOT EXISTS relations (
    id TEXT PRIMARY KEY,
    topicId TEXT NOT NULL,
    sourceEntityId TEXT,
    targetEntityId TEXT,
    relationType TEXT NOT NULL,
    description TEXT,
    confidence REAL,
    metadata TEXT,  -- JSONオブジェクト（検索用にインデックス追加を検討）
    organizationId TEXT,
    companyId TEXT,

    -- RAG検索用の追加カラム
    searchableText TEXT,  -- relationType + description + metadataの重要
フィールドを結合した検索用テキスト

    -- ChromaDB同期状態
    chromaSynced INTEGER DEFAULT 0,
    chromaSyncError TEXT,
    lastChromaSyncAttempt TEXT,

    -- タイムスタンプ
    createdAt TEXT NOT NULL,
    updatedAt TEXT NOT NULL,

    FOREIGN KEY (sourceEntityId) REFERENCES entities(id),
    FOREIGN KEY (targetEntityId) REFERENCES entities(id),
    FOREIGN KEY (organizationId) REFERENCES organizations(id),
    FOREIGN KEY (companyId) REFERENCES companies(id),
    CHECK ((organizationId IS NOT NULL AND companyId IS NULL) OR
           (organizationId IS NULL AND companyId IS NOT NULL))
);

-- 検索用インデックス
CREATE INDEX IF NOT EXISTS idx_relations_searchable_text ON
relations(searchableText);
CREATE INDEX IF NOT EXISTS idx_relations_organizationId ON
relations(organizationId);
CREATE INDEX IF NOT EXISTS idx_relations_topicId ON relations(topicId);
CREATE INDEX IF NOT EXISTS idx_relations_relationType ON
relations(relationType);
```

2. ChromaDBコレクション構造の最適化

コレクション名の修正

```
// organizationIdが空文字列の場合の処理を修正
let collection_name = if organization_id.is_empty() {
    "topics_all".to_string()  // または "topics_global"
```

```
} else {
    format!("topics_{}", organization_id)
};
```

メタデータの最適化

```
// 最小限のメタデータのみを保存
const metadata = {
  topicId: topic.topicId,
  meetingNoteId: topic.meetingNoteId,
  organizationId: topic.organizationId || 'all',
  title: topic.title.substring(0, 200),  // 長すぎる場合は切り詰め
  semanticCategory: topic.semanticCategory || '',
  // embeddingModel, embeddingVersionなどは削除（必要に応じて別テーブルに保存）
};
```

3. RAG検索結果の最適化

KnowledgeGraphSearchResultの拡張

```
export interface KnowledgeGraphSearchResult {
  type: SearchResultType;
  id: string;
  score: number;
  similarity: number;

  // エンティティの場合
  entity?: Entity;
  relatedRelations?: Relation[];  // 関連するリレーション
  relatedTopics?: TopicSummary[];  // 関連するトピック（要約）

  // リレーションの場合
  relation?: Relation;
  sourceEntity?: EntitySummary;  // ソースエンティティの要約
  targetEntity?: EntitySummary;  // ターゲットエンティティの要約
  topic?: TopicSummary;  // 関連するトピックの要約

  // トピックの場合
  topicId?: string;
  meetingNoteId?: string;
  topic?: TopicSummary;  // トピックの詳細情報（title, contentSummaryなど）
}

interface TopicSummary {
  topicId: string;
  title: string;
  contentSummary: string;  // contentの要約（200文字程度）
```

```
  semanticCategory?: string;
  keywords?: string[];
}

interface EntitySummary {
  id: string;
  name: string;
  type: string;
  displayName?: string;
}
```

4. AIアシスタント用コンテキストの最適化

コンテキスト生成の改善

```
export async function getKnowledgeGraphContext(
  queryText: string,
  limit: number = 5,
  filters?: {...},
  maxTokens: number = 3000
): Promise<string> {
  // 1. 検索結果を取得
  const results = await searchKnowledgeGraph(queryText, limit * 2,
filters);

  // 2. 優先度スコアでソート
  const prioritizedResults = results
    .map(r => ({
      ...r,
      priorityScore: calculatePriorityScore(r, queryText)
    }))
    .sort((a, b) => b.priorityScore - a.priorityScore);

  // 3. トークン制限内でコンテキストを構築
  const contextParts: string[] = [];
  let currentTokens = 0;

  for (const result of prioritizedResults) {
    const contextString = generateOptimizedContextString(result);
    const estimatedTokens = estimateTokens(contextString);

    if (currentTokens + estimatedTokens > maxTokens) {
      break;
    }

    contextParts.push(contextString);
    currentTokens += estimatedTokens;
  }

  return contextParts.join('\n\n');
```

```
}

function generateOptimizedContextString(
  result: KnowledgeGraphSearchResult
): string {
  const parts: string[] = [];

  if (result.type === 'entity' && result.entity) {
    parts.push(`**エンティティ: ${result.entity.name}**`);
    parts.push(`タイプ: ${result.entity.type}`);
    if (result.entity.displayName) {
      parts.push(`表示名: ${result.entity.displayName}`);
    }
    // 重要なメタデータのみ
    if (result.entity.metadata) {
      const importantFields = ['role', 'department', 'position'];
      for (const field of importantFields) {
        if (result.entity.metadata[field]) {
          parts.push(`${field}: ${result.entity.metadata[field]}`);
        }
      }
    }
    // 関連情報
    if (result.relatedRelations && result.relatedRelations.length > 0) {
      parts.push(`関連リレーション: ${result.relatedRelations.length}件`);
    }
    parts.push(`関連度: ${(result.score * 100).toFixed(1)}%`);
  } else if (result.type === 'relation' && result.relation) {
    parts.push(`**リレーション: ${result.relation.relationType}**`);
    if (result.relation.description) {
      parts.push(`説明: ${result.relation.description.substring(0,
200)}`);
    }
    if (result.sourceEntity) {
      parts.push(`ソース: ${result.sourceEntity.name}`);
    }
    if (result.targetEntity) {
      parts.push(`ターゲット: ${result.targetEntity.name}`);
    }
    parts.push(`関連度: ${(result.score * 100).toFixed(1)}%`);
  } else if (result.type === 'topic' && result.topic) {
    parts.push(`**トピック: ${result.topic.title}**`);
    if (result.topic.contentSummary) {
      parts.push(`内容: ${result.topic.contentSummary}`);
    }
    if (result.topic.semanticCategory) {
      parts.push(`カテゴリ: ${result.topic.semanticCategory}`);
    }
    parts.push(`関連度: ${(result.score * 100).toFixed(1)}%`);
  }

  return parts.join('\n');
}
```

実装優先度

高優先度（即座に実装）

1. **ChromaDBコレクション名の修正**: `organizationId`が空文字列の場合の処理
2. **RAG検索結果の拡張**: トピック情報（title, contentSummary）を含める
3. **コンテキスト生成の最適化**: トークン制限内で重要な情報のみを含める

中優先度（次期リリース）

1. **SQLiteテーブルの最適化**: `searchableText`カラムの追加とインデックス
2. **メタデータの最適化**: ChromaDBのメタデータから不要な情報を削除
3. **関連情報の取得**: エンティティ、リレーション、トピックの関連情報を取得

低優先度（将来的な改善）

1. **全文検索の最適化**: SQLiteのFTS（Full-Text Search）の導入
2. **キャッシュの最適化**: 検索結果のキャッシュ戦略の改善
3. **パフォーマンス最適化**: クエリの最適化とインデックスの追加

マイグレーション計画

フェーズ1: 非破壊的変更

1. 新しいカラムを追加（`searchableText`, `contentSummary`など）
2. 既存データを移行（バックグラウンドジョブ）
3. 新しいカラムにインデックスを追加

フェーズ2: アプリケーション側の変更

1. RAG検索結果の拡張
2. コンテキスト生成の最適化
3. ChromaDBコレクション名の修正

フェーズ3: クリーンアップ

1. 不要なカラムの削除（慎重に）
2. 古いデータの削除
3. パフォーマンステストと最適化

注意事項

1. **後方互換性**: 既存のデータとAPIとの互換性を保つ
2. **データ移行**: 既存データの移行を慎重に行う
3. **パフォーマンス**: インデックスの追加による書き込み性能への影響を考慮
4. **テスト**: 各フェーズで十分なテストを実施

データの扱いについて

データを空にする場合

メリット

- **マイグレーション処理が簡単**: 新しいスキーマでテーブルを作成するだけ
- **テストがしやすい**: クリーンな状態から始められる
- **古いデータの不整合がない**: 既存データの不整合を気にしなくて良い
- **開発速度が上がる**: データ移行処理を実装する必要がない

デメリット

- **既存データが失われる**: テストデータとして使えない
- **本番環境では使えない**: 本番データを失うことはできない
- **動作確認が難しい**: 既存データでの動作確認ができない

推奨される状況

- **開発環境**: テストデータが不要、または簡単に再生成できる
- **初期開発段階**: まだ本番環境にデプロイしていない
- **データが少ない**: 移行コストが高い

データを保持したままマイグレーションする場合

メリット

- **既存データを保持**: テストデータとして使える
- **本番環境でも安全**: 既存データを失わない
- **動作確認が容易**: 既存データでの動作確認ができる

デメリット

- **マイグレーション処理が複雑**: バックアップ→再作成→データ移行が必要
- **古いデータの不整合が残る可能性**: 既存データの不整合を修正する必要がある
- **実装時間がかかる**: データ移行処理の実装とテストが必要

推奨される状況

- **本番環境**: 既存データを失うことができない
- **テストデータが重要**: 既存データでテストしたい
- **データが多い**: 再生成が困難

推奨アプローチ

開発環境の場合

1. **データをバックアップ**: 念のためバックアップを取る

2. **データを空にする**: 新しいスキーマでクリーンに始める
3. **テストデータを再生成**: 必要に応じて新しい形式でテストデータを作成

本番環境の場合

1. **非破壊的マイグレーション**: 新しいカラムを追加するだけ（既存カラムは削除しない）
2. **データ移行**: バックグラウンドジョブで既存データを移行
3. **段階的移行**: フェーズごとに移行を進める

実装方針の選択

オプション1: データを空にする（推奨: 開発環境）

```
// 既存テーブルを削除して再作成
conn.execute("DROP TABLE IF EXISTS topics", [])?;
conn.execute("DROP TABLE IF EXISTS entities", [])?;
conn.execute("DROP TABLE IF EXISTS relations", [])?;

// 新しいスキーマでテーブルを作成
conn.execute("CREATE TABLE topics (...)", [])?;
// ...
```

オプション2: データを保持したままマイグレーション（推奨: 本番環境）

```
// 1. バックアップテーブルを作成
conn.execute("CREATE TABLE topics_backup AS SELECT * FROM topics", [])?;

// 2. 新しいカラムを追加（非破壊的）
conn.execute("ALTER TABLE topics ADD COLUMN searchableText TEXT", [])?;
conn.execute("ALTER TABLE topics ADD COLUMN contentSummary TEXT", [])?;

// 3. 既存データを移行（バックグラウンドジョブ）
// UPDATE topics SET searchableText = ... WHERE searchableText IS NULL;

// 4. インデックスを追加
conn.execute("CREATE INDEX idx_topics_searchable_text ON
topics(searchableText)", [])?;
```

判断基準

以下の質問に答えて、適切なアプローチを選択してください：

1. **環境**: 開発環境ですか？本番環境ですか？
2. **データの重要性**: 既存データは重要ですか？簡単に再生成できますか？
3. **データ量**: データはどのくらいありますか？（少ない/多い）
4. **テストの必要性**: 既存データでのテストは必要ですか？

推奨される手順（開発環境の場合）

1. **バックアップを取る**（念のため）

```
# SQLiteデータベースのバックアップ
cp database.db database_backup_$(date +%Y%m%d).db
```

2. **ChromaDBのバックアップ**（念のため）

```
# ChromaDBデータディレクトリのバックアップ
cp -r chromadb_data chromadb_data_backup_$(date +%Y%m%d)
```

3. **データを空にする**
    - SQLite: テーブルを削除して再作成
    - ChromaDB: コレクションを削除（必要に応じて）
4. **新しいスキーマでテーブルを作成**
5. **テストデータを再生成**（必要に応じて）

推奨される手順（本番環境の場合）

1. **非破壊的マイグレーション**: 新しいカラムを追加するだけ
2. **データ移行スクリプト**: 既存データを新しい形式に移行
3. **段階的移行**: フェーズごとに移行を進める
4. **ロールバック計画**: 問題が発生した場合のロールバック計画を準備