Land Database Adapter Documentation

Overview

The Land Database Adapter extends the PostgreSQL Database Adapter to provide specialized functionality for managing and querying virtual real estate data. It combines vector-based semantic search with structured metadata queries to enable powerful and flexible property searches.

Key Components

LandSearchParams Interface

```
interface LandSearchParams {
    neighborhoods?: string[];
    zoningTypes?: ZoningType[];
    plotSizes?: PlotSize[];
    buildingTypes?: BuildingType[];
    distances?: {
        ocean?: { maxMeters?: number; category?: DistanceCategory };
        bay?: { maxMeters?: number; category?: DistanceCategory };
    };
    building?: {
        floors?: { min?: number; max?: number };
        height?: { min?: number; max?: number };
    };
    rarity?: {
        rankRange?: { min?: number; max?: number };
    };
    coordinates?: {
        center: { x: number; y: number };
        radius: number;
    };
}
```

This interface defines all possible search parameters for property queries. All fields are optional, allowing for flexible search combinations.

Core Methods

createLandMemory

```
async createLandMemory(memory: LandPlotMemory): Promise<void>
```

Creates a new land property record in the database.

Parameters:

- `memory`: A LandPlotMemory object containing property details and metadata

Example:

```
await landDB.createLandMemory({
    id: "uuid",
    content: {
        text: "Luxury oceanfront property...",
        metadata: {
            neighborhood: "North Star",
            zoning: ZoningType.Residential,
            // ... other metadata
        }
    }
});
```

searchLandByMetadata

```
async searchLandByMetadata(params: LandSearchParams):
Promise<LandPlotMemory[]>
```

Searches for properties using metadata filters.

Parameters:

- `params`: Search criteria following the LandSearchParams interface

Example:

```
const luxuryProperties = await landDB.searchLandByMetadata({
    neighborhoods: ["North Star"],
    rarity: {
        rankRange: { max: 500 }  // Premium properties
    },
    distances: {
        ocean: { category: DistanceCategory.Close }
    }
});
```

searchLandByCombinedCriteria

```
async searchLandByCombinedCriteria(
    embedding: number[],
    metadata: Partial<LandSearchParams>,
    similarity_threshold: number = 0.7
): Promise<LandPlotMemory[]>
```

Combines semantic search with metadata filtering.

Parameters:

- `embedding`: Vector representation of search query
- `metadata`: Metadata filters
- `similarity_threshold`: Minimum similarity score (0-1)

Example:

```
const results = await landDB.searchLandByCombinedCriteria(
    queryEmbedding,
    {
        plotSizes: [PlotSize.Mega],
        buildingTypes: [BuildingType.Highrise]
    },
    0.8
);
```

getNearbyProperties

```
async getNearbyProperties(
    coordinates: { x: number; y: number },
    radiusMeters: number,
    limit: number = 10
): Promise<LandPlotMemory[]>
```

Finds properties within a specified radius.

Parameters:

- `coordinates`: Center point coordinates
- `radiusMeters`: Search radius in meters
- `limit`: Maximum number of results

Example:

```
const nearbyProps = await landDB.getNearbyProperties(
    { x: 500, y: 300 },
    1000,  // 1km radius
    5      // Top 5 results
);
```

updateLandMetadata

```
async updateLandMetadata(
    memoryId: UUID,
    metadata: Partial<LandPlotMetadata>
): Promise<void>
```

Updates specific metadata fields for a property.

Parameters:

- `memoryId`: Property identifier
- `metadata`: Partial metadata object with fields to update

Example:

```
await landDB.updateLandMetadata(propertyId, {
    rarity: {
        rank: 250,
        category: "Premium"
    }
});
```

Advanced Features

Spatial Search

The adapter supports spatial queries using PostgreSQL's geometric operations:

- Point-based distance calculations
- Radius-based proximity search
- Coordinate-based filtering

Combined Search Capabilities

- Semantic similarity using vector embeddings
- Metadata filtering with multiple criteria
- Spatial constraints
- Rarity and ranking filters

Query Optimization

- Efficient JSON field querying
- Index utilization for common search patterns
- Type casting for performance
- Array operations for bulk comparisons

Error Handling

The adapter includes comprehensive error handling:

- Input validation
- Type checking
- Query error recovery
- Logging through elizaLogger

Usage Best Practices

1. **Metadata Queries**

    - Use the most specific filters possible
    - Combine multiple criteria for precise results
    - Consider using ranges for numerical values

2. **Semantic Search**

    - Provide quality embeddings for better results
    - Adjust similarity threshold based on needs
    - Combine with metadata for better precision

3. **Spatial Queries**

    - Use appropriate radius values
    - Consider performance with large radius searches
    - Combine with other filters when possible

4. **Updates**

    - Use partial updates when possible
    - Validate metadata before updates
    - Consider embedding updates if text changes

Dependencies

- @ai16z/adapter-postgres
- @ai16z/eliza
- PostgreSQL with vector extension

Performance Considerations

1. **Index Usage**

    - JSON field indices for metadata queries
    - Vector indices for semantic search
    - Spatial indices for location queries

2. **Query Optimization**

    - Combined queries are executed efficiently
    - Metadata filtering happens at database level
    - Spatial calculations use native PostgreSQL functions

3. **Memory Management**

  - Large result sets are handled in chunks
  - Connection pooling for better resource usage
  - Proper cleanup of database resources