Natural Language to SQL in Elixir with MCP

Requirements Document

---

1. Project Overview

1.1 Purpose

This project aims to build a system that translates natural language queries into SQL queries, executes them against a PostgreSQL database, and visualizes the results using the Tucan library in Elixir. The system will leverage the Model Context Protocol (MCP) to facilitate communication between the language model and the database.

1.2 Scope

The system will:

- Accept natural language input through a web interface
- Parse and understand natural language queries
- Generate SQL queries based on user input
- Execute SQL queries against PostgreSQL
- Present query results as visualizations using Tucan
- Handle errors gracefully and provide feedback

1.3 User Stories

- As a user, I want to type questions in natural language so that I can query the database without knowing SQL.
- As a user, I want to see visualizations of my query results so that I can better understand the data.
- As a user, I want feedback when my query is unclear or cannot be processed.
- As a user, I want suggestions for refining my query when the initial attempt is unsuccessful.

---

2. Functional Requirements

2.1 Natural Language Input

- **FR-1.1:** The system shall provide a text input box for natural language queries.
- **FR-1.2:** The system shall support a variety of query types, including:
  - Simple selections (e.g., "Show me all employees")
  - Filtered queries (e.g., "Show employees in the marketing department")
  - Aggregated data (e.g., "What is the average salary of engineers?")

  - Sorted results (e.g., "List departments by size, largest first")
  - Limited results (e.g., "Show the top 5 highest-paid employees")
  - Temporal queries (e.g., "Show sales from last month")

2.2 Natural Language Understanding

- **FR-2.1:** The system shall parse natural language queries to identify:
  - Intent (select, aggregate, filter, sort)
  - Entities (tables, columns)
  - Conditions and filters
  - Limits and offsets
  - Sorting criteria
- **FR-2.2:** The system shall handle common synonyms for database entities (e.g., "staff" for "employees").
- **FR-2.3:** The system shall maintain context across consecutive queries to support follow-up questions.

2.3 SQL Generation and Execution

- **FR-3.1:** The system shall generate valid PostgreSQL SQL queries based on the parsed natural language.
- **FR-3.2:** The system shall execute generated SQL queries against the specified PostgreSQL database.
- **FR-3.3:** The system shall handle query errors gracefully and provide user-friendly error messages.
- **FR-3.4:** The system shall implement security measures to prevent SQL injection.
- **FR-3.5:** The system shall log all generated and executed SQL queries for debugging and auditing.

2.4 Data Visualization

- **FR-4.1:** The system shall automatically select appropriate visualization types based on query results.
- **FR-4.2:** The system shall support multiple visualization types using Tucan, including:
  - Bar charts
  - Line charts
  - Pie charts
  - Scatter plots
  - Tables for raw data
- **FR-4.3:** The system shall allow users to switch between different visualization types when appropriate.
- **FR-4.4:** The system shall ensure visualizations are correctly labeled with titles, axes, and legends.

2.5 User Interface

- **FR-5.1:** The system shall provide a clean, intuitive web interface.
- **FR-5.2:** The system shall display both the generated SQL and the visualization.
- **FR-5.3:** The system shall provide a history of past queries.
- **FR-5.4:** The system shall support basic query editing and re-execution.

---

3. Technical Requirements

3.1 Technology Stack

- **TR-1.1:** The system shall be built using Elixir and Phoenix Framework.
- **TR-1.2:** The system shall use Ecto for database interactions.
- **TR-1.3:** The system shall use PostgreSQL as the database system.
- **TR-1.4:** The system shall use Tucan for data visualization, which is built on VegaLite.
- **TR-1.5:** The system shall implement MCP for language model communication.
- **TR-1.6:** The system shall use HTTPoison for external API calls if using third-party language models.
- **TR-1.7:** The front-end shall use Phoenix LiveView for real-time interactions.

3.2 Model Context Protocol (MCP) Implementation

- **TR-2.1:** The system shall implement an MCP client using Hermes MCP or equivalent Elixir library.
- **TR-2.2:** The MCP implementation shall handle communication between the language model and the database.
- **TR-2.3:** The MCP server shall provide standardized endpoints for:
  - Query parsing
  - SQL generation
  - Schema introspection
  - Error handling

3.3 Database Requirements

- **TR-3.1:** The system shall connect to an existing PostgreSQL database.
- **TR-3.2:** The system shall support introspection of database schema to understand available tables and columns.
- **TR-3.3:** The system shall have read-only access to the database by default, with optional write capabilities.
- **TR-3.4:** The system shall handle large result sets efficiently, with pagination if necessary.

3.4 Performance Requirements

- **TR-4.1:** The system shall process natural language queries and return results within 5 seconds for simple queries.
- **TR-4.2:** The system shall handle multiple concurrent users.
- **TR-4.3:** The system shall implement caching for common queries and visualization results.

3.5 Security Requirements

- **TR-5.1:** The system shall sanitize all user inputs to prevent SQL injection.
- **TR-5.2:** The system shall implement authentication if required.
- **TR-5.3:** The system shall log all access and query attempts.
- **TR-5.4:** The system shall respect database access permissions.

---

4. System Architecture

4.1 High-Level Architecture

The system will follow a layered architecture:

1. **Presentation Layer** - Phoenix web interface
2. **Application Layer** - Business logic, natural language processing, and visualization logic
3. **Data Access Layer** - Database connection and query execution
4. **External Services Layer** - MCP and language model integration

4.2 Component Diagram

```
+-------------------+      +-------------------+      +-------------------+
|                   |      |                   |      |                   |
|  Web Interface    |----->|  Query Processor  |----->|  MCP Controller   |
|  (Phoenix)        |      |  (Elixir)         |      |  (Hermes MCP)     |
|                   |      |                   |      |                   |
+-------------------+      +-------------------+      +-------------------+
        ^                           |                          |
        |                           v                          v
        |                  +-------------------+      +-------------------+
        |                  |                   |      |                   |
        |                  |  SQL Generator    |<---->|  Language Model   |
        |                  |  (Elixir)         |      |  (External/Local) |
        |                  |                   |      |                   |
        |                  +-------------------+      +-------------------+
        |                           |
        |                           v
        |                  +-------------------+
        |                  |                   |
        |                  |  Database Access  |
        |                  |  (Ecto/Postgrex)  |
        |                  |                   |
        |                  +-------------------+
        |                           |
        |                           v
        |                  +-------------------+
        |                  |                   |
        |                  |  PostgreSQL DB    |
        |                  |                   |
        |                  +-------------------+
        |                           |
        |                           v
+-------------------+      +-------------------+
|                   |      |                   |
|  Visualization    |<-----|  Result Processor |
|  (Tucan/VegaLite) |      |  (Elixir)         |
|                   |      |                   |
+-------------------+      +-------------------+
```

4.3 Data Flow

1. User enters natural language query in web interface
2. Query processor receives the query and sends it to MCP controller
3. MCP controller communicates with language model to parse the query
4. SQL generator creates a valid SQL query based on the parsed intent

5. Database access layer executes the query against PostgreSQL
6. Result processor formats the raw query results
7. Visualization component generates appropriate visual representation
8. Web interface displays both SQL and visualization to the user

---

5. Database Schema

5.1 Schema Introspection

The system must be able to introspect and understand any existing PostgreSQL database schema. Key introspection points include:

- Tables and their relationships
- Column names, data types, and constraints
- Foreign key relationships
- Common query patterns

5.2 Sample Schema (for Testing)

For development and testing, a sample employees database with the following structure is recommended:

```
sql
```

```sql
-- Departments table
CREATE TABLE departments (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  location VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Positions table
CREATE TABLE positions (
  id SERIAL PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  level INTEGER,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Employees table
CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  phone VARCHAR(20),
  hire_date DATE NOT NULL,
  department_id INTEGER REFERENCES departments(id),
  position_id INTEGER REFERENCES positions(id),
  manager_id INTEGER REFERENCES employees(id),
  salary DECIMAL(10, 2),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Projects table
CREATE TABLE projects (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  description TEXT,
  start_date DATE,
  end_date DATE,
  budget DECIMAL(12, 2),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```sql
-- Employee-Project assignments (many-to-many)
CREATE TABLE employee_projects (
  id SERIAL PRIMARY KEY,
  employee_id INTEGER REFERENCES employees(id),
  project_id INTEGER REFERENCES projects(id),
  role VARCHAR(50),
  hours_allocated INTEGER,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  UNIQUE(employee_id, project_id)
);

-- Sample data inserts
INSERT INTO departments (name, location) VALUES
  ('Engineering', 'Building A'),
  ('Marketing', 'Building B'),
  ('Human Resources', 'Building A'),
  ('Sales', 'Building C'),
  ('Research', 'Building D');

-- Additional position and employee sample data
-- ...
```

---

6. Natural Language Processing

6.1 Query Parsing Approach

The system shall use one of the following approaches for natural language processing:

6.1.1 External LLM Integration

- Connect to an external language model API (OpenAI GPT, Anthropic Claude, etc.)
- Configure API parameters for optimal text-to-SQL conversion
- Handle rate limiting and fallbacks

6.1.2 Local Language Processing

- Implement pattern matching for common query types
- Use entity recognition for database schema elements
- Apply rule-based parsing for query construction

6.2 MCP Implementation

The MCP implementation will:

- Create standardized communication between the language model and database

- Define message formats for query parsing requests
- Handle response processing and error conditions
- Maintain session context for follow-up queries

6.3 Training and Improvement

- Log successful and failed queries for analysis
- Allow feedback on query translations
- Periodically update language patterns based on usage

---

7. Implementation Details

7.1 Project Structure

The recommended project structure follows standard Phoenix/Elixir conventions:

```
nlsql/
├── _build/
├── assets/
├── config/
├── deps/
├── lib/
│   ├── nlsql/
│   │   ├── application.ex
│   │   ├── repo.ex
│   │   ├── schema/
│   │   ├── nlp/
│   │   │   ├── parser.ex
│   │   │   ├── intent_extractor.ex
│   │   │   └── mcp_client.ex
│   │   ├── sql/
│   │   │   ├── generator.ex
│   │   │   ├── sanitizer.ex
│   │   │   └── executor.ex
│   │   └── viz/
│   │       ├── chart_builder.ex
│   │       └── tucan_adapter.ex
│   └── nlsql_web/
│       ├── controllers/
│       ├── live/
│       ├── templates/
│       ├── views/
│       ├── router.ex
│       └── endpoint.ex
├── priv/
│   ├── repo/
│   └── static/
├── test/
└── mix.exs
```

7.2 Key Modules and Functions

7.2.1 NLP Module

```elixir
defmodule NLSQL.NLP do
  @moduledoc """
  Natural language processing module for query parsing.
  """

  @doc """
  Parse a natural language query and extract structured information.
  """
  @spec parse_query(String.t()) :: {:ok, map()} | {:error, String.t()}
  def parse_query(query_text) do
    # Implementation
  end

  @doc """
  Maintain context between queries.
  """
  @spec with_context(String.t(), map()) :: {:ok, map()} | {:error, String.t()}
  def with_context(query_text, previous_context) do
    # Implementation
  end
end
```

7.2.2 MCP Client Module

```elixir
defmodule NLSQL.MCP.Client do
  @moduledoc """
  Model Context Protocol client implementation.
  """

  @doc """
  Initialize MCP client with configuration.
  """
  @spec init(map()) :: {:ok, pid()} | {:error, term()}
  def init(config) do
    # Implementation
  end

  @doc """
  Send a natural language query to the language model through MCP.
  """
  @spec process_query(pid(), String.t()) :: {:ok, map()} | {:error, term()}
  def process_query(client, query_text) do
    # Implementation
  end

  @doc """
  Provide database schema information to the language model.
  """
  @spec provide_schema(pid(), map()) :: :ok | {:error, term()}
  def provide_schema(client, schema_info) do
    # Implementation
  end
end
```

7.2.3 SQL Generator Module

```elixir
defmodule NLSQL.SQL.Generator do
  @moduledoc """
  SQL generation from structured query information.
  """

  @doc """
  Generate SQL from the parsed query intent.
  """
  @spec generate(map()) :: {:ok, String.t()} | {:error, String.t()}
  def generate(parsed_query) do
    # Implementation
  end

  @doc """
  Sanitize SQL input to prevent injection.
  """
  @spec sanitize(String.t()) :: String.t()
  def sanitize(sql) do
    # Implementation
  end
end
```

7.2.4 Database Module

elixir

```elixir
defmodule NLSQL.Database do
  @moduledoc """
  Database interaction functions.
  """

  @doc """
  Execute a SQL query.
  """
  @spec execute(String.t()) :: {:ok, list()} | {:error, term()}
  def execute(sql) do
    # Implementation
  end

  @doc """
  Introspect database schema.
  """
  @spec introspect_schema() :: {:ok, map()} | {:error, term()}
  def introspect_schema() do
    # Implementation
  end
end
```

7.2.5 Visualization Module

elixir

```elixir
defmodule NLSQL.Viz do
  @moduledoc """
  Visualization generation using Tucan.
  """

  @doc """
  Create appropriate visualization based on query results.
  """
  @spec visualize(list(), map()) :: {:ok, term()} | {:error, term()}
  def visualize(results, options) do
    # Implementation
  end

  @doc """
  Determine the best chart type for given data.
  """
  @spec recommend_chart_type(list()) :: atom()
  def recommend_chart_type(results) do
    # Implementation
  end
end
```

7.3 Phoenix LiveView Implementation

elixir

```elixir
defmodule NLSQLWeb.QueryLive do
  use NLSQLWeb, :live_view

  @impl true
  def mount(_params, _session, socket) do
    {:ok, assign(socket,
      query_text: "",
      sql: nil,
      results: nil,
      chart: nil,
      error: nil,
      history: []
    )}
  end

  @impl true
  def handle_event("submit", %{"query" => query_text}, socket) do
    # Process query and update socket assigns
    # Implementation
  end

  @impl true
  def handle_event("change_chart", %{"type" => chart_type}, socket) do
    # Change visualization type
    # Implementation
  end

  @impl true
  def render(assigns) do
    # LiveView template
    # Implementation
  end
end
```

---

8. Testing Requirements

8.1 Unit Tests

- **Test-1.1:** Test NLP parsing of various query types
- **Test-1.2:** Test SQL generation from parsed intents
- **Test-1.3:** Test database query execution
- **Test-1.4:** Test visualization generation

- **Test-1.5:** Test MCP client functionality

8.2 Integration Tests

- **Test-2.1:** Test end-to-end flow from natural language to visualization
- **Test-2.2:** Test error handling at various stages
- **Test-2.3:** Test context maintenance between queries
- **Test-2.4:** Test handling of ambiguous queries

8.3 Performance Tests

- **Test-3.1:** Test response time for various query complexities
- **Test-3.2:** Test concurrent user handling
- **Test-3.3:** Test memory usage with large result sets

8.4 Test Data

- Create a comprehensive set of test queries covering different query patterns
- Generate realistic dataset for employee database
- Prepare expected SQL outputs for comparison

---

9. Deployment Considerations

9.1 Environment Setup

- **Dep-1.1:** Elixir 1.14+ runtime
- **Dep-1.2:** Phoenix 1.7+
- **Dep-1.3:** PostgreSQL 13+
- **Dep-1.4:** Node.js 16+ (for Phoenix assets)

9.2 Configuration Management

- **Dep-2.1:** Database connection configuration
- **Dep-2.2:** MCP server configuration
- **Dep-2.3:** Language model API keys (if applicable)
- **Dep-2.4:** Logging configuration

9.3 Monitoring and Logging

- **Dep-3.1:** Log all queries and their translations
- **Dep-3.2:** Track success/failure rates
- **Dep-3.3:** Monitor performance metrics

- **Dep-3.4:** Implement error alerting

---

10. Future Enhancements

10.1 Potential Improvements

- **Enh-1.1:** Support for more complex query types (joins, window functions)
- **Enh-1.2:** User query templates and favorites
- **Enh-1.3:** Query explanation in natural language
- **Enh-1.4:** Custom visualization options
- **Enh-1.5:** Support for multiple database dialects
- **Enh-1.6:** Voice input for natural language queries
- **Enh-1.7:** Export options for visualizations and data

---

11. Glossary

- **MCP**: Model Context Protocol - A standardized way to connect AI models to different data sources and tools.
- **NLP**: Natural Language Processing - The field of computer science focused on parsing and understanding human language.
- **SQL**: Structured Query Language - The standard language for interacting with relational databases.
- **Tucan**: An Elixir library for data visualization built on top of VegaLite.
- **VegaLite**: A high-level grammar of interactive graphics built on Vega.
- **Phoenix**: A web development framework for Elixir.
- **Ecto**: A database wrapper and query generator for Elixir.
- **LiveView**: A Phoenix feature that enables real-time, server-rendered HTML.

Appendix A: Sample Queries

The system should handle queries like:

- "Show me all employees"
- "List departments and their employee count"
- "Who are the top 5 highest-paid employees?"
- "Show the average salary by department"
- "How many employees were hired in each year?"
- "What is the total salary expense per department?"

- "Which department has the highest average salary?"
- "Show employees who manage more than 3 people"
- "List projects with budget over $100,000"
- "Show departments with more than 10 employees"

Appendix B: Resources

- Elixir Documentation: https://elixir-lang.org/docs.html
- Phoenix Framework: https://hexdocs.pm/phoenix/overview.html
- Ecto Documentation: https://hexdocs.pm/ecto/Ecto.html
- PostgreSQL Documentation: https://www.postgresql.org/docs/
- Tucan Documentation: (Reference the appropriate documentation URL)
- Hermes MCP Documentation: https://hexdocs.pm/hermes_mcp/readme.html
- Model Context Protocol: https://mcp.so/

Document Version: 1.0
Creation Date: 2025-05-05
Author: Claude AI Assistant