AI Engineering Assignment: NLP Query Engine for Employee Data

Problem Statement

Build a natural language query system for an employee database that dynamically adapts to the actual schema and can handle both structured employee data and unstructured documents. The system should work without hard-coding table names, column names, or relationships.

Core Challenge

Create a query engine that:

- Automatically discovers the database schema when connected
- Handles natural language queries without knowing column names in advance
- Optimizes performance for production use
- Works with any reasonable employee database structure

Technical Requirements

1. User Interface Requirements

Build a web interface that provides:

Data Ingestion Panel:

```python
# Required endpoints
POST /api/ingest/database    # Connect to database and discover schema
POST /api/ingest/documents   # Upload documents (bulk upload supported)
GET  /api/ingest/status      # Check ingestion progress
```

Query Interface:

```python
# Query endpoint
POST /api/query              # Process natural language query
GET  /api/query/history      # Get previous queries (for caching demo)
```

Required UI Features:

- Database connection form (connection string, test connection)
- Document upload (drag-and-drop, multiple files, progress indicator)
- Query input box with auto-suggestions

- Results display (tables for structured data, cards for documents)
- Performance metrics display (response time, cache hit/miss)
- Schema visualization (show discovered tables and relationships)

Example Interface Flow:

```
1. User provides database connection string
2. System shows discovered schema (tables, columns, relationships)
3. User uploads documents (PDFs, CSVs, etc.)
4. System shows processing progress
5. User enters natural language query
6. System displays results with source attribution
```

2. Dynamic Schema Discovery

Instead of hard-coding table/column names, your system must:

```python
class SchemaDiscovery:
    def analyze_database(self, connection_string: str) -> dict:
        """
        Connect to database and automatically discover:
        - Table names and their likely purpose (employees, departments, etc.)
        - Column names and data types
        - Relationships between tables
        - Sample data for context understanding

        Should work with variations like:
        - employee, employees, emp, staff
        - salary, compensation, pay
        - dept, department, division
        """
        pass

    def map_natural_language_to_schema(self, query: str, schema: dict) -> dict:
        """
        Map user's natural language to actual database structure.
        Example: "salary" in query → "compensation" in database
        """
        pass
```

2. Query Processing Pipeline

Build a production-ready pipeline that:

```python
class QueryEngine:
    def __init__(self, connection_string: str):
        # Auto-discover schema on initialization
        self.schema = SchemaDiscovery().analyze_database(connection_string)
        self.cache = QueryCache()

    def process_query(self, user_query: str) -> dict:
        """
        Process natural language query with:
        - Query classification (SQL vs document search vs hybrid)
        - Caching for repeated queries
        - Performance optimization
        - Error handling and fallbacks
        """
        pass

    def optimize_sql_query(self, sql: str) -> str:
        """
        Optimize generated SQL:
        - Use indexes when available
        - Limit result sets appropriately
        - Add pagination for large results
        """
        pass
```

3. Document Processing

Handle employee documents efficiently:

```python
```

```python
class DocumentProcessor:
    def process_documents(self, file_paths: list) -> None:
        """
        Process multiple document types:
        - Auto-detect file type (PDF, DOCX, TXT, CSV)
        - Choose optimal chunk size (don't hard-code to 512 tokens)
        - Generate embeddings in batches for efficiency
        - Store with proper indexing for fast retrieval
        """
        pass

    def dynamic_chunking(self, content: str, doc_type: str) -> list:
        """
        Intelligent chunking based on document structure:
        - Resumes: Keep skills and experience sections together
        - Contracts: Preserve clause boundaries
        - Reviews: Maintain paragraph integrity
        """
        pass
```

4. Production Features

- **Concurrent Users**: Handle at least 10 simultaneous users
- **Response Time**: 95% of queries under 2 seconds
- **Error Recovery**: Graceful handling of database connection issues
- **Query Validation**: Prevent SQL injection and dangerous queries
- **Monitoring**: Log query performance and errors

User Interface Implementation

Required Web Application

Build a full-stack web application with the following interfaces:

1. Data Ingestion Interface

```
python
```

```
# Backend API endpoints
@app.post("/api/connect-database")
async def connect_database(connection_string: str):
    """
    Connect to database and auto-discover schema
    Return: discovered tables, columns, relationships
    """

@app.post("/api/upload-documents")
async def upload_documents(files: List[UploadFile]):
    """
    Accept multiple document uploads
    Process and store with embeddings
    Return: processing status and document IDs
    """

@app.get("/api/ingestion-status/{job_id}")
async def get_status(job_id: str):
    """
    Return progress of document processing
    """
```

2. Query Interface

```python
@app.post("/api/query")
async def process_query(query: str):
    """
    Process natural language query
    Return: results, query_type, performance_metrics, sources
    """

@app.get("/api/schema")
async def get_schema():
    """
    Return current discovered schema for visualization
    """
```

3. Frontend Requirements

Create a web interface (React, Vue, or plain HTML/JS) with:

```javascript
```

```
// Main components needed:

// 1. Connection Panel
<DatabaseConnector>
  - Input: Database connection string
  - Button: Test Connection
  - Display: Success/Error message
  - Visualization: Show discovered schema as tree/graph
</DatabaseConnector>

// 2. Document Upload Panel
<DocumentUploader>
  - Drag-and-drop zone for files
  - Support: PDF, DOCX, TXT, CSV
  - Progress bar for each file
  - Batch upload capability
  - Show: Processing status, extracted text preview
</DocumentUploader>

// 3. Query Interface
<QueryPanel>
  - Search input with auto-complete
  - Query history dropdown
  - Submit button
  - Display: Loading spinner during processing
</QueryPanel>

// 4. Results Display
<ResultsView>
  - For SQL: Table view with pagination
  - For Documents: Card view with highlighted matches
  - For Hybrid: Combined view with source labels
  - Show: Response time, cache hit indicator
  - Export: Download results as CSV/JSON
</ResultsView>

// 5. Metrics Dashboard
<MetricsDashboard>
  - Current schema stats (tables, documents)
  - Query performance graph
  - Cache hit rate
  - Active connections
</MetricsDashboard>
```

4. UI/UX Requirements

- Clean, professional interface
- Responsive design (mobile-friendly)
- Real-time feedback for all operations
- Error messages that guide users
- Dark/light mode (bonus)

Example User Flow

```
1. Landing Page
   └── Two main sections: "Connect Data" and "Query Data"

2. Connect Data Flow:
   ├── Database Tab:
   │   ├── Enter: postgresql://user:pass@localhost/company_db
   │   ├── Click: "Connect & Analyze"
   │   └── Shows: Discovered 5 tables, 47 columns, 3 relationships
   │
   └── Documents Tab:
       ├── Drag: 50 employee resumes
       ├── Shows: Processing... 23/50 complete
       └── Complete: "50 documents indexed successfully"

3. Query Data Flow:
   ├── Input: "Show me all Python developers in Engineering"
   ├── System shows: "Searching database and documents..."
   └── Results:
       ├── Table: 5 employees from database
       ├── Documents: 3 resumes mentioning Python
       └── Metrics: Query took 1.3s (cache miss)
```

5. Performance Optimizations

Required optimizations:

- **Query Caching**: Cache frequent queries with intelligent invalidation
- **Connection Pooling**: Reuse database connections
- **Batch Processing**: Process multiple embeddings together
- **Async Operations**: Non-blocking I/O for better concurrency
- **Result Pagination**: Handle large result sets efficiently

5. Production Features

- **Concurrent Users**: Handle at least 10 simultaneous users

- **Response Time**: 95% of queries under 2 seconds
- **Error Recovery**: Graceful handling of database connection issues
- **Query Validation**: Prevent SQL injection and dangerous queries
- **Monitoring**: Log query performance and errors

Test Scenarios

Your system will be tested with different database schemas (structure not provided in advance):

Schema Variation 1:

```sql
employees (emp_id, full_name, dept_id, position, annual_salary, join_date, office_location)
departments (dept_id, dept_name, manager_id)
```

Schema Variation 2:

```sql
staff (id, name, department, role, compensation, hired_on, city, reports_to)
documents (doc_id, staff_id, type, content, uploaded_at)
```

Schema Variation 3:

```sql
personnel (person_id, employee_name, division, title, pay_rate, start_date)
divisions (division_code, division_name, head_id)
```

Your system should handle ALL variations without code changes.

Query Examples to Support

Basic Queries

- "How many employees do we have?"
- "Average salary by department"
- "List employees hired this year"
- "Who reports to John Smith?"

Complex Queries

- "Top 5 highest paid employees in each department"
- "Employees with Python skills earning over 100k"

- "Show me performance reviews for engineers hired last year"
- "Which departments have the highest turnover?"

Edge Cases

- Ambiguous queries: "Show me John" (multiple Johns)
- Missing data: "Salary for contractors" (when contractors not in DB)
- Complex aggregations: "Month-over-month hiring trends"

Deliverables

1. GitHub Repository

Source Code:

- Backend API (FastAPI/Flask/Django)
- Frontend application (React/Vue/HTML+JS)
- Database migration scripts
- Document processing pipeline
- No hard-coded schemas or assumptions
- README with clear setup instructions

Project Structure:

```
project/
├── backend/
│   ├── api/
│   │   ├── routes/
│   │   │   ├── ingestion.py
│   │   │   ├── query.py
│   │   │   └── schema.py
│   │   ├── services/
│   │   │   ├── schema_discovery.py
│   │   │   ├── document_processor.py
│   │   │   └── query_engine.py
│   │   └── models/
│   └── main.py
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   │   ├── DatabaseConnector.js
│   │   │   ├── DocumentUploader.js
│   │   │   ├── QueryPanel.js
│   │   │   └── ResultsView.js
│   │   └── App.js
│   └── public/
├── docker-compose.yml
├── requirements.txt
├── package.json
└── README.md
```

Deliverables

1. GitHub Repository

Source Code:

- Backend API (FastAPI/Flask/Django)
- Frontend application (React/Vue/HTML+JS)
- Database migration scripts
- Document processing pipeline
- No hard-coded schemas or assumptions
- README with clear setup instructions

Project Structure:

```
project/
├── backend/
│   ├── api/
│   │   ├── routes/
│   │   │   ├── ingestion.py
│   │   │   ├── query.py
│   │   │   └── schema.py
│   │   ├── services/
│   │   │   ├── schema_discovery.py
│   │   │   ├── document_processor.py
│   │   │   └── query_engine.py
│   │   └── models/
│   └── main.py
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   │   ├── DatabaseConnector.js
│   │   │   ├── DocumentUploader.js
│   │   │   ├── QueryPanel.js
│   │   │   └── ResultsView.js
│   │   └── App.js
│   └── public/
├── requirements.txt
├── package.json
└── README.md
```

Configuration:

```yaml
# config.yml - All configuration should be external
database:
  connection_string: ${DATABASE_URL}
  pool_size: 10

embeddings:
  model: "sentence-transformers/all-MiniLM-L6-v2"
  batch_size: 32

cache:
  ttl_seconds: 300
  max_size: 1000
```

Testing:

- Unit tests with mocked schemas

- Integration tests demonstrating adaptability
- Performance benchmarks

2. Loom Video Demo (5-7 minutes)

Required demonstrations:

1. **Data Ingestion** (2 min)
   - Connect to a database and show schema discovery
   - Upload multiple documents and show processing
   - Display ingestion status and success metrics
2. **Query Interface** (3 min)
   - Execute 6 different queries via the UI:
     - 2 SQL queries (show table results)
     - 2 document queries (show relevant chunks)
     - 2 hybrid queries (show combined results)
   - Demonstrate cache hits for repeated queries
   - Show error handling for invalid queries
3. **Performance & Features** (2 min)
   - Run concurrent queries from different browser tabs
   - Show response time metrics on the interface
   - Demonstrate schema visualization
   - Export results functionality

Evaluation Criteria

Core Functionality (35 points)

- **User Interface (10 pts)**
  - Intuitive data ingestion flow
  - Clean query interface
  - Results presentation clarity
  - Error handling in UI
- **Schema Discovery (10 pts)**
  - Accurate detection of tables and relationships
  - Handling naming variations
  - No hard-coding
- **Query Processing (10 pts)**

    - Correct query classification
    - Accurate SQL generation
    - Relevant document search results
- **Data Ingestion (5 pts)**
    - Multiple file format support
    - Progress tracking
    - Batch processing capability

Performance & Optimization (30 points)

- **Speed (10 pts)**
    - Query response time < 2 seconds
    - Efficient use of caching
- **Scalability (10 pts)**
    - Concurrent query handling
    - Resource efficiency
- **Optimization Techniques (10 pts)**
    - Connection pooling
    - Batch processing
    - Async operations where appropriate

Code Quality (20 points)

- **Architecture (10 pts)**
    - Clean, maintainable code
    - Proper separation of concerns
    - Good abstractions
- **Production Readiness (10 pts)**
    - Comprehensive error handling
    - Logging and monitoring
    - Security considerations

Documentation & Presentation (10 points)

- **Documentation (5 pts)**
    - Clear setup instructions
    - Well-documented code
- **Demo Quality (5 pts)**

  - Clear presentation
  - Covers all requirements

Constraints

- Python 3.8+ (FastAPI, Flask, or Django)
- PostgreSQL or MySQL (SQLite acceptable for demo)
- Open-source models only (no paid API requirements for evaluation)
- Should run on 8GB RAM machine

Common Pitfalls to Avoid

- Hard-coding table or column names
- Assuming fixed schema structure
- Loading entire datasets into memory
- Not handling connection failures
- Ignoring SQL injection risks
- Single-threaded blocking operations

FAQs

Q: Can I use LLMs like ChatGPT or Claude for query processing? A: You can use free-tier APIs (OpenAI GPT-3.5, Google Gemini, etc.) or open-source models. Ensure your solution works within free tier limits. The evaluator should not need paid API keys.

Q: What if I can't implement vector search perfectly? A: Focus on a working solution. Simple keyword matching with good engineering is better than a complex but broken vector search. Partial credit is given for functional alternatives.

Q: Should I implement authentication for the web interface? A: No, authentication is not required. Focus on the core functionality of ingestion and querying.

Q: How should I handle very large files? A: Implement reasonable limits (e.g., 10MB per file) and show appropriate error messages. Mention scalability considerations in your README.

Q: Can I use ORMs like SQLAlchemy? A: Yes, any Python libraries are acceptable. Using ORMs for database abstraction is encouraged for cleaner code.

Q: What frontend framework should I use? A: Use whatever you're comfortable with - React, Vue, vanilla JavaScript, or even server-side rendered templates (Jinja2, etc.). The UI functionality matters more than the framework choice.

Q: How sophisticated should the schema discovery be? A: At minimum, detect tables, columns, and explicit foreign keys. Bonus points for inferring implicit relationships or detecting naming patterns.

Q: Should I implement real-time updates? A: Not required. Polling for ingestion status is sufficient. WebSockets are optional.

Q: How many test cases should I include? A: Include at least 5-10 unit tests covering critical functions and 2-3 integration tests showing the system works end-to-end.

Q: Can I use pre-trained embeddings or must I train my own? A: Use pre-trained models like sentence-transformers. Training custom models is not expected.

Q: What if my solution doesn't handle all edge cases? A: Document known limitations in your README. We value transparency and understanding of trade-offs over claiming perfect solutions.

Q: How should I handle queries that span multiple tables? A: Support basic JOINs where relationships exist. Complex multi-table queries can return a "query too complex" message with suggestions to simplify.

Q: Is caching mandatory? A: Yes, implement at least basic query result caching to demonstrate understanding of performance optimization.

Q: Should the UI be production-ready? A: No, a functional interface is sufficient. Focus on usability over aesthetics. Bootstrap or basic CSS is fine.

Q: Can I use a different database like MongoDB? A: The assignment requires SQL databases since schema discovery and SQL generation are key evaluation points. Stick to PostgreSQL, MySQL, or SQLite.

Submission

1. Complete the implementation
2. Record Loom video demonstration
3. Push code to public GitHub repository
4. Submit via Google Form: [Submission Link]
    - GitHub repository URL
    - Loom video URL
    - Technical documentation (PDF)
    - Performance benchmark results

Deadline: [To be specified]

Questions? Contact: [recruiter email]