Web Application to Convert User Input to SQL and Run it on Databases

Group Information

Group Name: Lazy_GCD

Team Members:

- Aritra Maji (22CS30011)
- Sayandeep Bhowmick (22CS10069)
- Agniva Saha (22CS10003)
- Ritabrata Bharati (22CS10059)
- Raaja Das (22CS30043)

Problem Statement

The project aims to develop a sophisticated web application that leverages Language Learning Models (LLMs) to convert natural language questions into SQL queries. Users can interact with PostgreSQL databases using plain English, with the system automatically generating and executing appropriate SQL queries based on the user's input. The application processes these natural language queries through an advanced RAG (Retrieval Augmented Generation) system, optimizing query generation for complex database operations.

Methodology

1. System Architecture

The application implements a modern three-tier microservices architecture with clear separation of concerns:

1.1 Frontend Architecture (Client Tier)

1. **Technology Stack**:

   - React 18 with Vite for optimized development and build performance
   - TailwindCSS with custom configuration for consistent styling
   - Material-UI components with theme customization
   - Framer-motion for fluid page transitions and micro-interactions
   - Redux Toolkit with custom middleware for state management
   - Axios with request/response interceptors for API communication
   - React Router v6 for declarative routing
   - React Query for server state management

2. **Key Components**:

   - Authentication components with Firebase and JWT integration
   - Database connection manager with real-time status updates
   - Advanced query builder with syntax highlighting
   - Interactive metadata visualization components
   - Real-time error boundary system

- Toast notification system
- Responsive layout engine with dark mode support
- Excel export functionality

1.2 Backend Architecture (Application Tier)

1. **Core Technologies**:

- Django REST Framework with custom viewsets and serializers
- PostgreSQL with connection pooling
- Firebase Admin SDK for OAuth integration
- JWT authentication with refresh token rotation

2. **Service Layers**:

- Multi-provider authentication service
- Database connection pool manager
- Query execution engine with transaction support
- Metadata extraction service
- File export service with multiple formats

1.3 LLM and RAG Integration

1. **Natural Language Processing**:

- Context-aware query understanding: Utilizes advanced NLP techniques to parse and comprehend user queries while maintaining context across multiple interactions within a session. The system analyzes query intent, entities, and relationships to ensure accurate interpretation.
- Schema-based query validation: Validates the parsed query against the database schema to ensure all referenced tables and columns exist. Performs type checking and relationship validation to prevent invalid queries.
- SQL syntax optimization: Implements intelligent query optimization by analyzing join paths, subqueries, and index usage. The system restructures complex queries for better performance while maintaining semantic equivalence.
- Error correction and suggestions: Provides real-time error detection and correction for common natural language mistakes. Offers intelligent suggestions for ambiguous terms and automatically resolves common query patterns.
- Query performance analysis: Monitors query execution time, resource usage, and result set size. Uses PostgreSQL's EXPLAIN ANALYZE to optimize query plans and suggest improvements.
- Natural language response generation: Converts technical SQL results back into natural language responses that are easy to understand. Includes context-aware explanations of the query logic.
- Context window management: Maintains conversation history and context across multiple queries in a session. Uses sliding window approach to manage memory efficiently while preserving relevant context.

2. **RAG Implementation**:

- GPU-accelerated processing with CUDA optimization: Leverages NVIDIA CUDA cores for parallel processing of language models and vector operations. Implements batched inference for optimal GPU utilization and reduced latency.

- Streaming RAG with dynamic batching: Processes large datasets efficiently through streaming architecture. Dynamically adjusts batch sizes based on available GPU memory and query complexity.
- Efficient JSON vectorization: Converts database schema and metadata into optimized vector representations. Uses custom encoding schemes to minimize memory footprint while preserving semantic relationships.
- Context-aware processing pipeline: Maintains and updates context through a sophisticated pipeline that tracks database state, user session, and query history. Enables accurate query understanding across complex interactions.
- Memory-efficient token management: Implements sliding window token management to handle large context windows. Prioritizes recent and relevant information while gracefully handling token limitations.
- Error recovery with fallback mechanisms: Provides robust error handling with multiple fallback options. Automatically switches to alternative models or approaches when primary processing fails.
- Real-time performance monitoring: Tracks and analyzes system performance metrics including latency, throughput, and resource utilization. Provides insights for optimization and scaling decisions.
- SSH tunneling with keepalive: Maintains secure, persistent connections to GPU servers through encrypted SSH tunnels. Implements automatic reconnection and session persistence mechanisms.

2. Core Features Implementation

2.1 Authentication System

1. **Firebase Integration**:

- Google OAuth 2.0 with custom scopes
- Email/password with strong validation
- Custom token generation with claims
- Session management with Redis
- Real-time auth state synchronization
- Password reset with secure OTP
- Token refresh with sliding window
- Multi-factor authentication support
- Rate limiting for auth attempts

2. **Custom JWT Authentication**:

- Token-based system with RSA signing
- Secure password hashing with bcrypt
- Redis-based session management
- Token blacklisting with TTL
- Role-based access control matrix
- CORS with preflight handling
- Cross-tab authentication sync
- Automatic token refresh
- Session timeout handling

2.2 Database Management System

1. **Connection Management**:

- Dynamic connection pooling with PgBouncer
- Encrypted connection strings
- SSL/TLS with certificate validation
- Connection health monitoring
- Automatic failover handling
- Connection pool optimization
- Load balancing support
- Query timeout management

2. **Metadata Management**:

- Schema extraction with dependencies
- Relationship inference engine
- AI-powered schema documentation
- Real-time schema tracking
- Version control integration
- Search with fuzzy matching
- Caching with invalidation
- Change tracking system

3. **Query Processing System**:

- Natural language parser with error recovery
- Context-aware SQL generation
- Query validation with explain analyze
- Parameterized query support
- Results pagination system
- Export pipeline architecture
- Query optimization suggestions
- Performance metrics collection

2.3 Natural Language to SQL Conversion

1. **Query Understanding**:

- Context extraction module: Analyzes the user's natural language query to identify key entities, relationships, and operations. Maintains session context to understand references to previous queries and results.
- Intent classification system: Determines the type of operation being requested (select, update, join, aggregate, etc.). Uses advanced NLP models to categorize query intent with high accuracy.
- Entity recognition engine: Identifies database objects (tables, columns, values) mentioned in the natural language query. Maps colloquial terms to their corresponding database entities.
- Schema matching algorithm: Matches identified entities against the database schema using fuzzy matching and semantic similarity. Handles aliases and common variations in table/column names.

- Ambiguity resolution: Resolves ambiguous references by considering schema relationships and query context. Implements intelligent disambiguation using schema metadata and usage patterns.
- Query complexity analysis: Evaluates query complexity to optimize processing approach. Determines join paths, subquery requirements, and potential performance impacts.
- Type inference system: Automatically infers data types for conditions and comparisons. Ensures type compatibility in generated SQL queries.
- Query validation rules: Applies comprehensive validation rules to ensure generated queries are valid and safe. Prevents SQL injection and malformed queries.

2. **SQL Generation**:

- Template-based generation: Utilizes a library of SQL templates based on query patterns and intent classification. Templates are dynamically populated with entities and conditions from the natural language analysis.
- Dynamic query construction: Builds SQL queries incrementally by combining validated components. Handles complex query structures including joins, subqueries, and aggregations.
- Schema-aware optimization: Leverages database schema information to optimize table joins and column selections. Considers indexes and foreign key relationships for efficient query paths.
- Join path optimization: Determines the most efficient join paths between tables based on relationship metadata. Minimizes unnecessary table scans and optimizes join conditions.
- Subquery handling: Manages nested queries and complex data relationships. Determines when to use subqueries versus joins for optimal performance.
- Error prevention system: Implements multiple validation layers to catch potential issues before query execution. Includes syntax validation, semantic checks, and security scanning.
- Performance optimization: Analyzes generated queries using PostgreSQL's query planner. Suggests improvements and restructures queries for better execution plans.
- Query explanation generation: Creates natural language explanations of generated SQL queries. Helps users understand the transformation from natural language to SQL.

3. **Query Execution**:

- Transaction management: Implements ACID-compliant transaction handling with automatic commit and rollback mechanisms. Ensures data consistency across complex operations.
- Error handling with rollback: Provides comprehensive error detection and recovery system. Automatically rolls back failed transactions to maintain database integrity.
- Result set processing: Efficiently handles large result sets through pagination and streaming. Implements memory-efficient data transformation for different output formats.
- Memory management: Uses optimized buffer management for large queries. Implements resource cleanup and connection pooling to prevent memory leaks.
- Connection pooling: Maintains an efficient pool of database connections using PgBouncer. Dynamically scales connection pool based on workload.

- Query timeout handling: Implements configurable timeout mechanisms for long-running queries. Provides graceful termination and cleanup of timed-out operations.
- Performance monitoring: Tracks query execution metrics including duration, resource usage, and cache effectiveness. Generates performance reports for optimization.
- Cache management: Implements multi-level caching strategy for frequently accessed data. Uses Redis for distributed caching with automatic invalidation.

4. **RAG Pipeline**:

- Query vectorization: Converts natural language queries into high-dimensional vector representations using advanced embedding techniques. Optimizes for semantic similarity matching with the knowledge base.
- Context retrieval system: Implements efficient retrieval of relevant context from the knowledge base using vector similarity search. Utilizes indexes for fast nearest neighbor lookups.
- Prompt engineering: Dynamically constructs prompts by combining user queries with retrieved context and database schema information. Optimizes prompt structure for maximum relevance.
- Response generation: Uses a multi-stage generation process that combines retrieved information with LLM capabilities. Ensures responses are both accurate and contextually appropriate.
- Quality assurance checks: Implements automated validation of generated responses against database schema and business rules. Includes syntax checking and semantic validation.
- Performance optimization: Uses batched processing and caching strategies to minimize latency. Implements adaptive batch sizing based on system load and request patterns.
- Error recovery: Provides robust error handling with automatic retry mechanisms. Includes fallback options for different failure scenarios.
- Streaming response: Implements efficient streaming of large result sets using pagination and chunked transfer. Optimizes memory usage while maintaining responsiveness.

3. User Interface Implementation

1. **Database Dashboard**:

- Real-time status monitoring
- Interactive connection manager
- Quick action system with shortcuts
- Error notification system
- Performance metrics visualization
- Activity logging system
- Permission management interface
- Bulk operation support

2. **Metadata Manager**:

- Interactive schema visualizer
- Relationship diagram generator
- Dynamic ER diagram system
- Advanced search functionality

- Schema comparison tools
- Export system with formats
- Version tracking interface
- Custom view support

3. **Query Interface**:

- Natural language input system
- SQL preview with highlighting
- Interactive query builder
- Results visualization
- Export functionality
- Query history tracking
- Performance analysis
- Error suggestion system

4. Security Implementation

1. **Authentication Security**:

- Token encryption system
- Password security policies
- Session management
- CORS configuration
- Rate limiting rules

2. **Database Security**:

- Connection encryption
- Query sanitization
- Access control matrix
- SSL/TLS configuration

3. **Application Security**:

- Input validation
- XSS prevention
- CSRF protection

Results and Implementation Overview

Core Functionality Achievements

1. **Query Processing**:

- Successfully implemented natural language understanding
- Accurate SQL query generation
- Efficient query execution
- Comprehensive error handling
- Performance optimization
- Result formatting
- Export capabilities

2. **Authentication**:

- Dual authentication system
- Secure session management
- Password recovery
- Role management
- Activity tracking
- Security monitoring
- Audit logging

3. **Database Operations**:

- Multiple database support
- Metadata management
- Query execution
- Export functionality
- Performance tracking
- Error recovery
- Backup systems

4. **User Interface**:

- Responsive design
- Dark mode support
- Interactive features
- Real-time updates
- Error handling
- Performance metrics
- Accessibility support

Deployment Details

Overview

This documentation outlines the deployment setup for a full-stack web application using Docker containers on a local machine with Cloudflare Tunnel for secure public access.

Deployment Infrastructure

- **Container Orchestration** : Docker Compose
- **Backend Server** : Gunicorn
- **Frontend Server** : Nginx
- **Public Access** : Cloudflare Tunnel

Container Architecture

1. **Frontend Container**
    - Serves React application through Nginx
    - Port: 3737:80

2. **Backend Container**
    - Runs Django application via Gunicorn
    - Port: 8000:8000
    - Manages static/media files through volumes
3. **Cloudflared Container**
    - Establishes secure tunnel to Cloudflare
    - Enables public access without port forwarding
    - IPv4/IPv6 agnostic

Key features

- **Security :** Hosted locally with secure Cloudflare Tunnel access
- **Persistence** : Docker volumes for database, static and media files
- **Environment Management :** Uses .env file for configuration
- **Container Dependencies :** Properly ordered startup with depends_on

Notes

- Backend API requests are automatically proxied through Nginx
- Static and media files are persisted through Docker volumes

Conclusion and Future Scope

Achievements

1. Implemented advanced natural language to SQL conversion
2. Built secure and scalable authentication system
3. Created intuitive and responsive interface
4. Integrated comprehensive database management
5. Implemented efficient RAG system
6. Developed robust metadata management
7. Established strong security measures

Future Scope

1. **Enhanced LLM Capabilities**:

- Multiple model support
- Advanced optimization
- Context learning
- Performance enhancements
- Language support
- Custom training
- Error handling

2. **Database Support**:

- Additional databases

- Advanced connections
- Enhanced metadata
- Backup systems
- Migration tools
- Performance tuning
- Monitoring systems

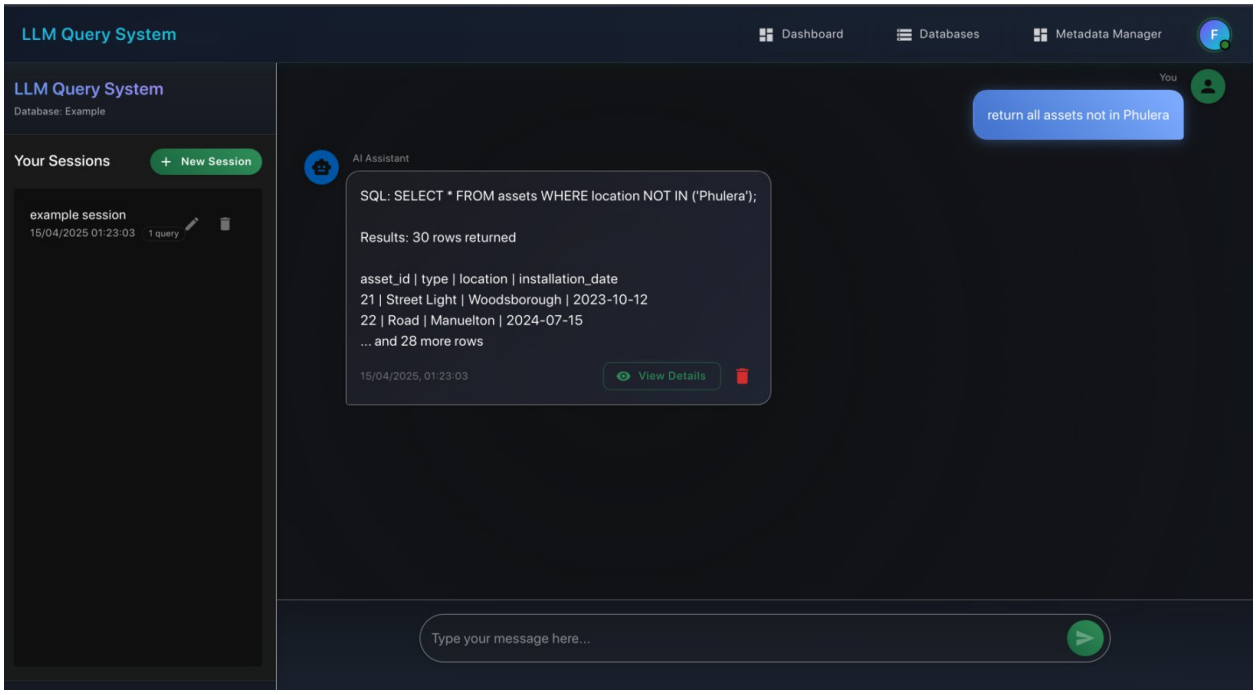
3. **User Experience**:

- Advanced visualizations
- Collaboration features
- Mobile applications
- Offline capabilities
- Accessibility
- Performance optimization
- Real-time analytics

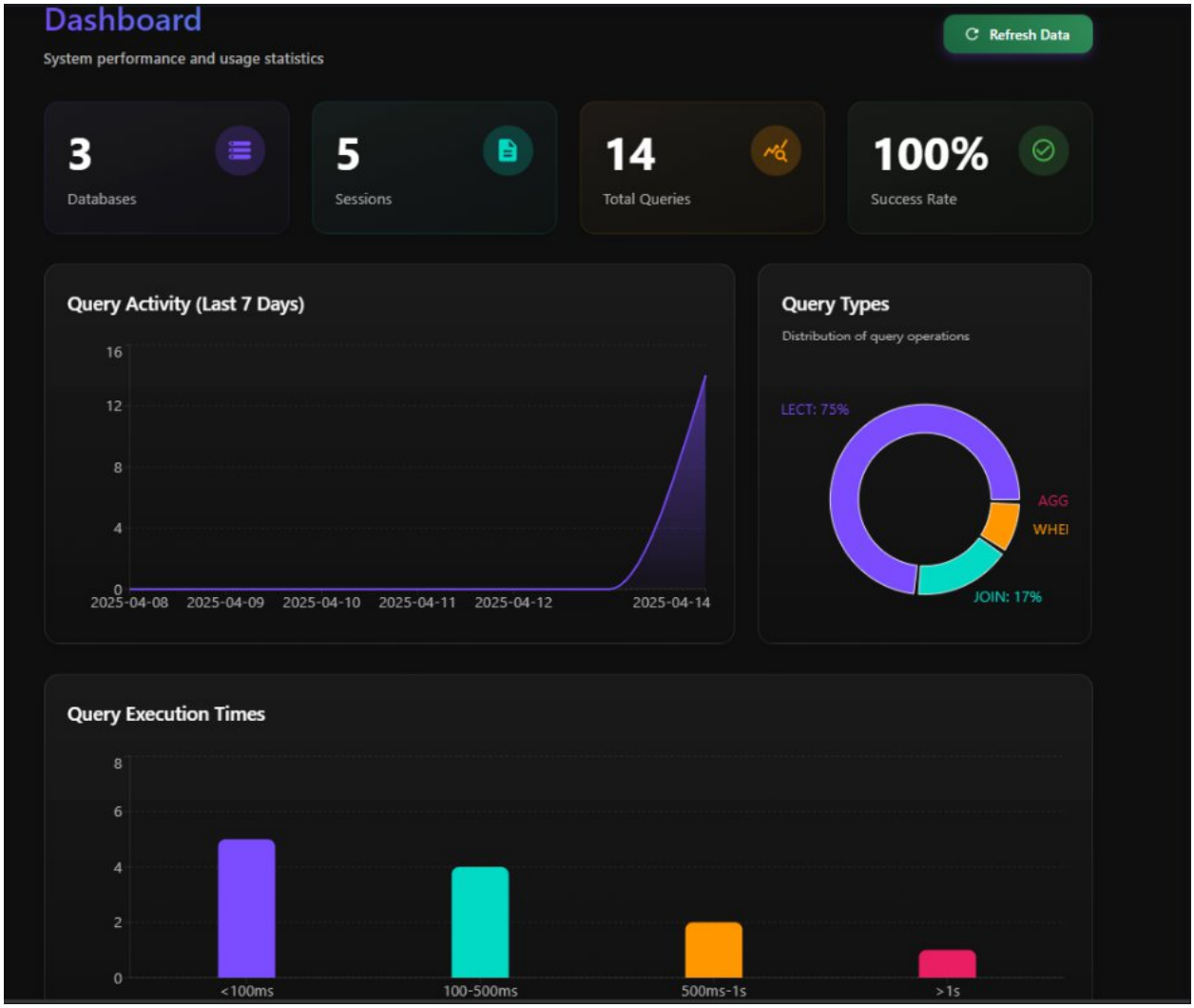
References

1. React Documentation (https://reactjs.org/)
2. Django REST Framework (https://www.django-rest-framework.org/)
3. Firebase Documentation (https://firebase.google.com/docs)
4. Material-UI Documentation (https://mui.com/)
5. TailwindCSS Documentation (https://tailwindcss.com/)
6. PostgreSQL Documentation (https://www.postgresql.org/docs/)
7. Redux Documentation (https://redux.js.org/)
8. Framer Motion Documentation (https://www.framer.com/motion/)

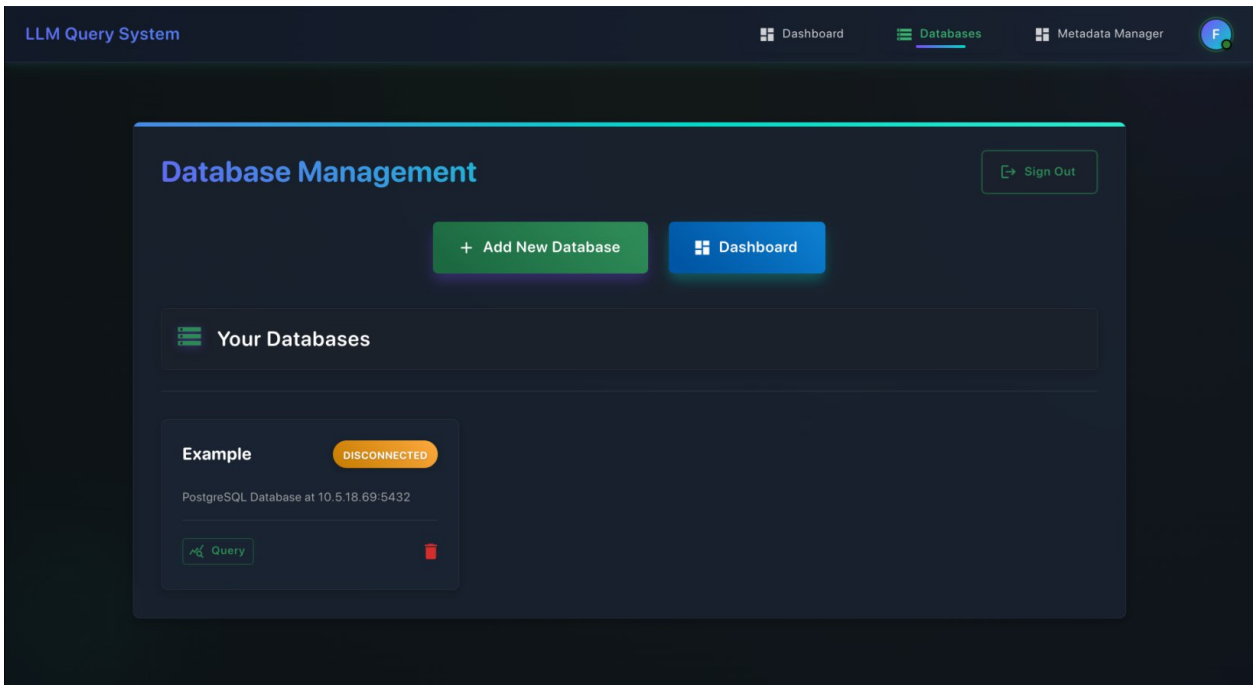
Sample Screenshots of our application



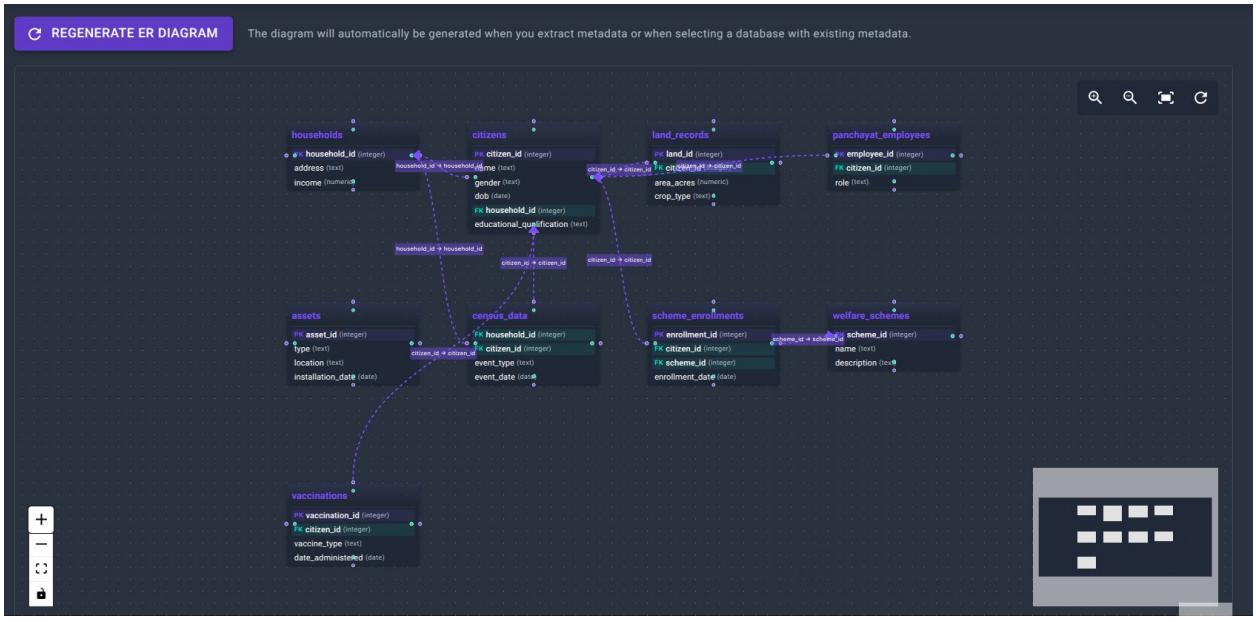
Query interface



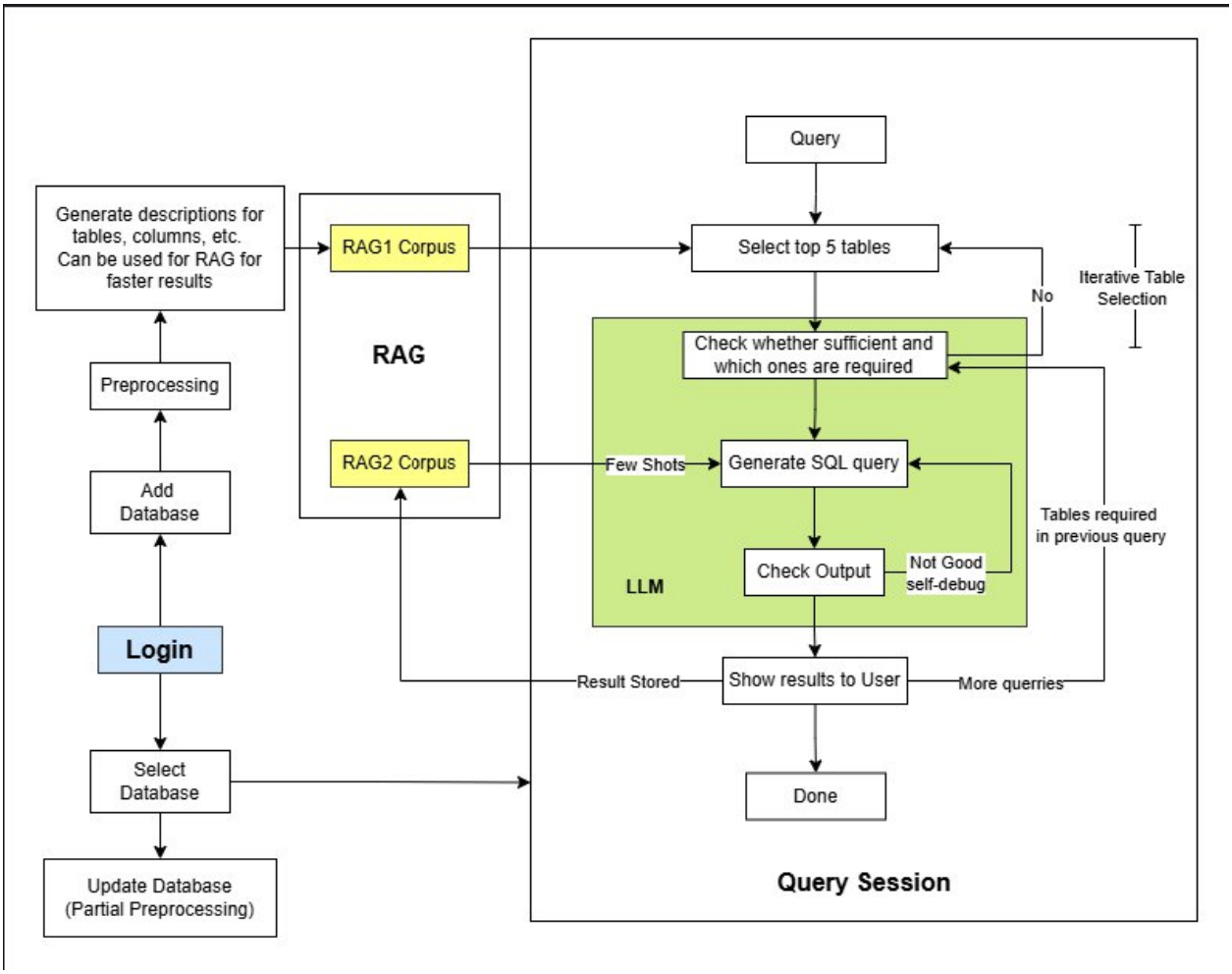
Dashboard



Database page



Auto-generated ER diagram



Our pipeline