**Title: LLM-Based Test Generation Framework for Relational Databases**

### **1. Why This Project?**

Relational databases power critical systems in domains like finance, healthcare, and e-commerce. While they are optimized for performance, even minor changes or enhancements can introduce subtle bugs that impact accuracy, security, or privacy.

This project aims to leverage Large Language Models (LLMs) to automate the generation of realistic test data and intelligent SQL test queries. Beyond crash testing, the tool will validate whether query results are logically correct. This shifts testing from mere survival to correctness assurance.

### **2. Target Users**

This tool is intended for:

* **Database engine developers** ensuring correctness through optimization cycles.
* **QA engineers** validating functional and non-functional correctness of database behavior.
* **Researchers** exploring innovative database testing strategies.

**Industries:** Especially valuable for privacy-critical domains such as banking, healthcare, and government systems.

### **3. System Overview**

#### **Core Features:**

**LF10 - Smart Table Generator**

* Uses LLMs to generate fake but realistic data given schema (tables, fields, types, keys).
* Honors constraints like primary/foreign keys and uniqueness.
* Use CONSTRAINT to define the ranges of the data in tables.
* Also ensure that Cardinality can be controlled.
* The distribution of numerical and textual fields should be analyzed.

**LF20 & LF30 - Query Generator with Oracle**

* Generated queries must be related to or query the tables generated.
* Generates meaningful SQL queries for the synthetic tables that test SQL functions like GROUP BY, comparison operators, ORDER AS etc.
* Also computes expected query results (oracle) for correctness validation.

**LF40 - Bug Spotter**

* Uses one or more of the classical techniques to spot the bugs:  
  + Metamorphic testing: Input variations yield predictable output variations.
  + Differential testing: Compare outputs across database versions.
  + Constraint-based testing: Push edge-case scenarios.

**LF50 - Bug Evaluator**

* Validates the tool on known faulty PostgreSQL builds.
* Demonstrates detection of subtle bugs missed by traditional testing.

**LF60 - Security Validator (Optional requirement)**

* Detects unsafe query constructs and patterns:  
  + SQL injection vectors
  + Data leakage via improperly escaped joins
  + Role-based permission escalation
* Uses LLM-generated adversarial queries and static security rules.

### **4. Inputs and Outputs**

**Inputs:**

| **Code** | **Input Type** | **Description** |
| --- | --- | --- |
| LD10 | Schema | Table names, fields, datatypes, constraints |
| LD20 | Data Profile | Desired value distributions, cardinality, etc. |

**Outputs:**

| **Output Type** | **Description** |
| --- | --- |
| Table Data | Generated synthetic records conforming to schema |
| SQL Queries | LLM-generated test queries |
| Oracle Results | Expected outputs for each query |
| Using table data, SQL queries and Oracle results, the following outputs will be generated:- | |
| Bug Reports | Detected mismatches or logical errors |
| Security Reports | Detected vulnerabilities or unsafe patterns |

### **5. Performance Targets**

The goal is to compare all the parameters with the state-of-the-art.

| **Code** | **Metric** | **Goal** |
| --- | --- | --- |
| LL10 | Table Generation Time | A reasonable table generation time. |
| LL20 | Query Evaluation Time | Validate query correctness within a reasonable time. |
| LL30 | Bug Detection Coverage | Identify both the logical/crash bugs |
| LL40 | Security Detection Rate | Detect SQL vulnerabilities |

### **6. Quality Goals**

* **Reliability:** Consistent generation of valid, schema-compliant data and queries.
* **Usability:** CLI and optional web UI for defining schemas and initiating test runs.
* **Efficiency:** Results returned in a reasonable time.
* **Extensibility (optional):** Easy addition of new test strategies and support for other DBMSs.
* **Security Awareness (optional):** Early detection of exploitable patterns in SQL logic.

### **7. Additional Considerations**

* Initial focus on **PostgreSQL**; future support for MySQL, SQLite, etc.
* All LLM prompts and outputs logged for reproducibility and transparency.
* All generated data must be obviously fake and privacy-safe.
* Community-supported security test packs will evolve over time.

### **8. Test Plan**

We will test the framework against known buggy PostgreSQL builds. We will evaluate:

* Crash detection capability.
* Logical bug identification using oracle results.
* Coverage of different SQL features and edge cases.
* Accuracy of security issue detection (e.g., SQL injection, unsafe joins).

### **9. Glossary**

* **LLM:** Large Language Model (e.g., GPT-4).
* **Oracle:** The expected result of a given SQL query.
* **Metamorphic Testing:** Verify outputs change predictably with input changes.
* **Differential Testing:** Compare behavior across two database systems.
* **Constraint-Based Testing:** Generate inputs that test edge-case logic.
* **SQL Injection:** A common security vulnerability where user input alters SQL logic.

### **10. Phases of Development**

| **Phase** | **Description** |
| --- | --- |
| 1 (MVP) | Synthetic Data generator. |
| 2 | Development of Query Generator |
| 3 | Add automatic oracle generation + test strategies |
| 4 | Evaluation and Reporting |
| 5 | Security test suite with auto vulnerability |

### **11. Requirements**

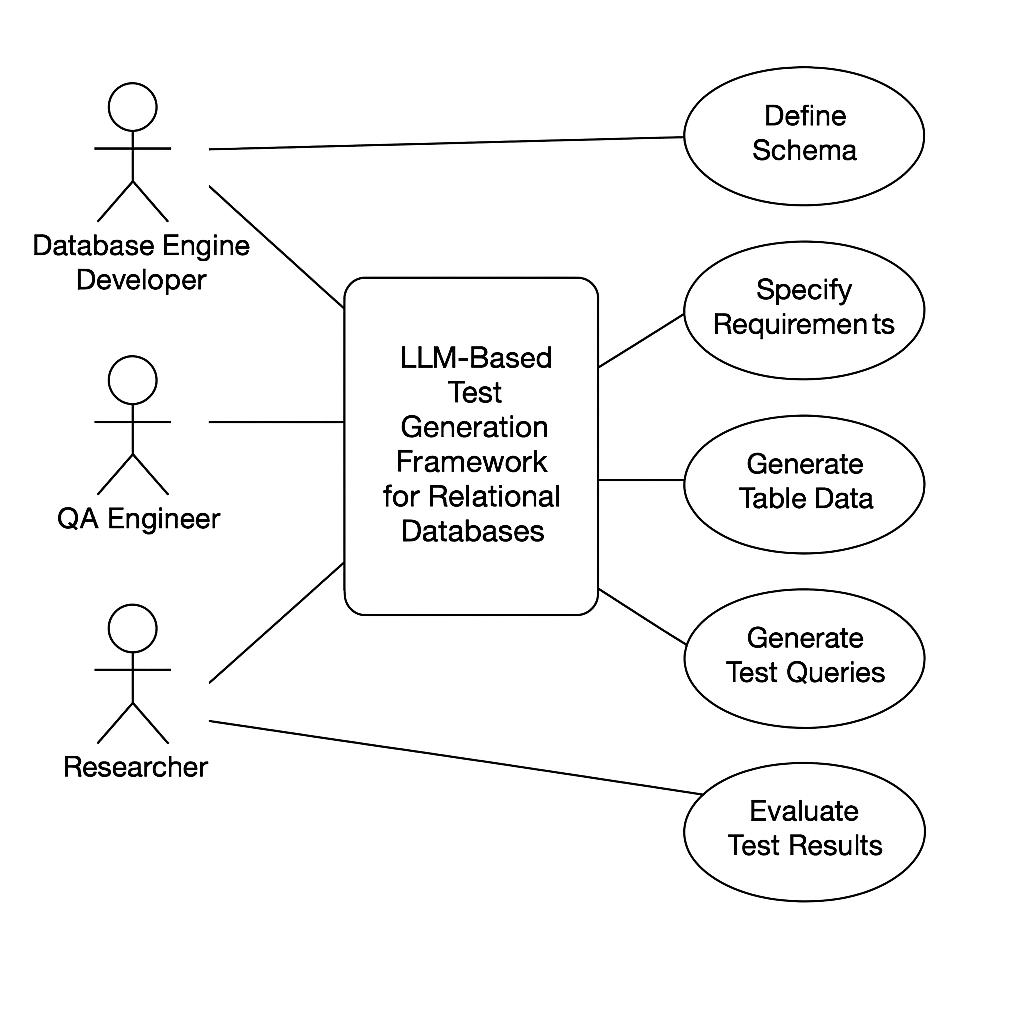
#### **✅ Functional Requirements**

| **Code** | **Requirements** |
| --- | --- |
| **Data Generation Component** | |
| FR1 | The system shall generate synthetic tabular data for PostgreSQL databases. |
| FR2 | The generated data must:  Match provided database schemas (tables, columns, data types).  Respect constraints such as Primary Keys, Foreign Keys, and NOT NULL.  Support realistic value distributions (e.g., normal, uniform distributions). |
| FR3 | The system should allow the user to specify:  Number of rows to generate.  Data value ranges and formats. |
| **Query Generation Component** | |
| FR4 | The system shall automatically generate SQL queries for testing PostgreSQL |
| FR5 | It must support:  Simple queries (SELECT, WHERE).  Complex queries (JOINs, GROUP BY, Aggregations).  Edge case queries to trigger corner scenarios (e.g., division by zero, NULL handling). |
| FR6 | The system shall use LLMs to generate queries based on provided schemas |
| **Oracle Generation and Testing Component** | |
| FR7 | The system shall implement strategies to solve the oracle problem, using one or more of the following:  Metamorphic Testing: Create logically equivalent queries and compare results.  Differential Testing: Compare query results across different PostgreSQL versions or other databases.  Constraint-Based Checking: Validate results using schema constraints. |
| **Evaluation and Reporting** | |
| FR8 | The system shall evaluate PostgreSQL’s behavior using generated queries and datasets. |
| FR9 | It should detect both crash bugs and logic bugs. |
| FR10 | It should provide reports summarizing found issues and inconsistencies. |
| FR11 | It should support testing for privacy and security-related bugs. |

#### **🔄 Non-Functional Requirements**

| **Code** | **Requirement** |
| --- | --- |
| NFR1 | The system should be modular, allowing easy extension to other databases if needed. |
| NFR2 | Should support both CLI and web UI modes |
| NFR3 | Should allow configurable test strategies and parameters |
| NFR4 | The tool should generate data and queries in reasonable time even for large schemas |
| NFR5 | The tool should ensure that sensitive or personal data is not used; only synthetic data is generated. |

**Use Case Diagram**

****