

This approach gives us a solid foundation before we begin coding. Do you want me to focus on any specific aspect of this solution as we move forward?

PS

ok so what you think can be the most optimal solution for this

Based on my analysis, the most optimal solution would be a layered approach that balances thoroughness with performance and safety. Here's what I recommend:

Optimal MariaDB Query Validator Solution

1. Two-Tier Validation System:
 - Static Analysis: Metadata-based validation without execution
 - Controlled Execution: Testing in a sandboxed environment
2. Core Components: a. Schema Cache:

- Load and cache database metadata at startup (tables, columns, types)
 - Refresh periodically or on schema changes
 - Use for fast lookup during validation
3. b. SQL Parser Integration:
- Integrate a robust SQL parser library (JSqlParser or similar)
 - Extract table/column references and validate against schema cache
 - No need to build a complex parser from scratch
4. c. Prepare Statement Validation:
- Use MariaDB's own parser via PreparedStatement
 - Catches syntax errors with high accuracy
5. d. Sandbox Execution Environment:
- Clone of production schema in a test database

- Read-only user permissions where possible
- Transaction wrapping with automatic rollback

6. Validation Flow:

- Parse query to extract objects and classify query type
- Validate object names against schema cache
- Check syntax using prepared statements
- For data-modifying queries, run in sandbox with EXPLAIN
- Return detailed validation report

7. Spring Boot Integration:

- REST API for manual validation
- Integration hooks for CI/CD pipelines
- Admin console for monitoring and configuration

8. Performance Optimizations:

- In-memory schema caching

- Validation result caching for repeated queries
- Parallel validation for multiple queries
- Incremental validation for large procedures

This solution provides:

- Fast validation for simple queries via schema cache
- Thorough validation for complex queries via controlled execution
- Protection against execution side effects
- Spring Boot integration for easy deployment
- Extensibility for future enhancements

The key advantage is avoiding the need to build a complete SQL parser while leveraging the database's own capabilities to validate syntax and semantics accurately.

Would you like me to start designing the core components of this solution in more detail?

PS

so you know how to do all these things

Yes, I have the knowledge to implement this solution. The approach leverages several key technologies and techniques:

1. Schema metadata retrieval: Using MariaDB's INFORMATION_SCHEMA tables to query database structure
2. SQL parsing: Using libraries like JSqlParser to extract table and column references without building a parser from scratch
3. JDBC/Spring integration: Leveraging Spring Boot's database connectivity for prepared statements and transaction management

4. Safe execution strategies: Implementing read-only transactions with rollback guarantees

The most challenging aspects are:

1. SQL parsing for complex queries - this is why I recommend using an established parser library rather than custom regex-based solutions
2. Handling MariaDB-specific syntax extensions, which would require thorough testing against various query types
3. Balancing validation thoroughness with performance, especially for large stored procedures

I can guide you through implementing any part of this solution - from setting up the initial Spring Boot project structure, to integrating SQL parsing, to implementing the validation logic and sandbox execution environment.

Would you like me to focus on a specific aspect of the implementation to start with?

PS

okay i told u earlier about my antlr project thta parser i have ready with me

That's excellent news! Having an ANTLR parser ready significantly simplifies the implementation. With your parser in place, we can focus on building the validation logic and integration with Spring Boot.

Let me outline how we can proceed using your existing ANTLR parser:

Implementation Approach with Your ANTLR Parser

1. Parser Integration:

- Use your existing ANTLR parser to parse SQL queries
- Extract table references, column references, and query structure

- This eliminates the need for a third-party SQL parsing library
- 2. Schema Validation Service:
 - Create a service that loads and caches schema metadata from MariaDB
 - Compare the parsed query's object references against this cache
 - Validate that all referenced tables and columns exist
- 3. Type Checking Module:
 - For INSERT/UPDATE operations, extract value expressions
 - Use the schema cache to verify data type compatibility
 - Flag potential type conversion issues
- 4. Query Preparation Validator:
 - Use JDBC PreparedStatement for syntax validation
 - Leverage the database engine's native parsing capabilities

5. Spring Boot Integration:

- Create REST endpoints for validation requests
- Design response format with detailed validation results
- Add configuration options for validation strictness

Simplified Implementation Structure

```
com.yourcompany.queryvalidator/  
├── config/  
│   └── DatabaseConfig.java  
├── controller/  
│   └──  
QueryValidatorController.java  
├── model/  
│   ├── ValidationResult.java  
│   ├── SchemaObject.java  
│   └── ValidationError.java  
├── parser/  
│   └── SQLParserIntegration.java  
(integrates your ANTLR parser)
```

```
|— service/
|   |— SchemaService.java
|   |
|   |—
|   |   QueryValidationService.java
|   |   |
|   |   |—
|   |   |   TypeValidationService.java
|   |   |
|   |   |— util/
|   |   |   |— QueryExecutionUtil.java
```

Next Steps

1. Schema Metadata Service: Create the service to load and cache database schema information
2. Parser Integration: Build the bridge between your ANTLR parser and the validation logic
3. Validation Rules: Implement the core validation logic using the parser output
4. API Layer: Design the REST interface for validation requests

5. Testing: Create comprehensive tests with various query types

Would you like me to provide more detailed implementation for any of these components? With your ANTLR parser already handling the SQL parsing complexity, we can focus on building the validation logic and Spring Boot integration.

PS

let me clarify you one thing the parser tool i created checks only for the syntax errors only

Thank you for clarifying that. This is an important distinction - your ANTLR parser checks for syntax errors only, but our validator needs to handle semantic validation as well (like verifying table and column names exist).

In this case, we'll need to build on top of your syntax parser to add schema validation capabilities. Here's how we can approach this:

Enhanced Solution Architecture

1. Layer 1: Syntax Validation (Your ANTLR Parser)
 - Use your existing parser to verify syntax correctness
 - This is the first check any query must pass
2. Layer 2: Schema Validation (New Component)
 - Query INFORMATION_SCHEMA to build a metadata cache
 - Extract object references from syntactically valid queries
 - Validate these references against the metadata cache

3. Layer 3: Semantic Validation (New Component)

- Check data type compatibility
- Validate foreign key constraints
- Analyze execution plans

Let's outline how these components would work together:

```
Query → Syntax Validation →  
Schema Extraction → Metadata  
Validation → Type Checking →  
Result
```

Implementation Approach

Since your parser only handles syntax, we need to add the capability to extract object references from valid SQL. We have two options:

1. Enhance your ANTLR parser to also extract table and column references

2. Build a separate extractor that works with the parse tree your ANTLR grammar produces

For the second approach, we'd need to:

1. Create Listeners/Visitors for your ANTLR grammar that extract:
 - Table references (FROM, JOIN, UPDATE, etc.)
 - Column references (SELECT, WHERE, etc.)
 - Data manipulation operations (INSERT values, UPDATE sets)
2. Build the Schema Validation Service:
 - Cache database metadata using INFORMATION_SCHEMA queries
 - Compare extracted references against this cache

Given this clarification, would you prefer to:

1. Enhance your existing ANTLR parser to extract object references?
2. Build a separate component that analyzes the parse tree from your parser?
3. Use a hybrid approach with your parser for syntax and JDBC metadata for schema validation?

Building a MariaDB Query Validator: Step-by-Step Guide

Here's a comprehensive guide to build your query validator tool using the hybrid approach with your ANTLR parser for syntax checking and JDBC metadata for schema validation.

Phase 1: Project Setup

1. Create a Spring Boot Project
 - Use Spring Initializr (<https://start.spring.io/>)
 - Add dependencies: Spring Web, Spring JDBC, MariaDB Driver
 - Download and import into your IDE
2. Configure Database Connection
 - Update `application.properties`:
properties

```
spring.datasource.url=jdbc:mariadb://localhost:3306/yourdb
spring.datasource.username=your_username
```

```
spring.datasource.password=your_password
```

- ```
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
```

### Integrate Your ANTLR Parser

- Add your ANTLR parser jar as a dependency

Create a service to interface with it

## Phase 2: Core Components Implementation

### Step 1: Create Basic Models

Create the following model classes:

1. `ValidationResult.java`

java

```
public class ValidationResult {
 private boolean valid = true;
 private List<ValidationError> errors = new ArrayList<>();
 private List<String> warnings = new ArrayList<>();

 // Getters, setters, and helper methods
 public void addError(ValidationError error) {
 this.valid = false;
 this.errors.add(error);
 }

 public void addWarning(String warning) {
 this.warnings.add(warning);
 }
}
```

`ValidationError.java`

java

```
public class ValidationError {
 public enum ErrorType {
 SYNTAX,
 UNKNOWN_TABLE,
 UNKNOWN_COLUMN,
 DATA_TYPE_MISMATCH,
 OTHER
 }
}
```



```

 }

 private ErrorType type;
 private String message;
 private int lineNumber = -1;
 private int position = -1;

 // Constructors, getters, setters
}

```

## SqlQuery.java

```

java
public class SqlQuery {
 private String sql;
 private int lineNumber;

 // Constructors, getters, setters

 1. }

```

## Step 2: Create the Schema Service

```

java
@Service
public class SchemaMetadataService {
 private final JdbcTemplate jdbcTemplate;

 // Schema cache data structures
 private Map<String, Set<String>> tableColumnMap = new
HashMap<>();
 private Map<String, Map<String, String>> columnTypeMap = new
HashMap<>();

 @Autowired
 public SchemaMetadataService(JdbcTemplate jdbcTemplate) {
 this.jdbcTemplate = jdbcTemplate;
 refreshSchemaCache();
 }

 @Scheduled(fixedRateString =
"${schema.cache.refresh.interval:3600000}")
 public void refreshSchemaCache() {
 loadTableAndColumnData();
 }
}

```

```

private void loadTableAndColumnData() {
 // Clear existing cache
 tableColumnMap.clear();
 columnTypeMap.clear();

 // Query for table and column information
 String sql = "SELECT TABLE_NAME, COLUMN_NAME, DATA_TYPE,
COLUMN_TYPE " +
 "FROM INFORMATION_SCHEMA.COLUMNS " +
 "WHERE TABLE_SCHEMA = DATABASE()";

 List<Map<String, Object>> results =
jdbcTemplate.queryForList(sql);

 for (Map<String, Object> row : results) {
 String tableName = ((String)
row.get("TABLE_NAME")).toLowerCase();
 String columnName = ((String)
row.get("COLUMN_NAME")).toLowerCase();
 String dataType = (String) row.get("DATA_TYPE");
 String columnType = (String) row.get("COLUMN_TYPE");

 // Add to table-column map
 tableColumnMap
 .computeIfAbsent(tableName, k -> new HashSet<>())
 .add(columnName);

 // Add to column-type map
 columnTypeMap
 .computeIfAbsent(tableName, k -> new HashMap<>())
 .put(columnName, columnType);
 }
}

public boolean tableExists(String tableName) {
 return tableColumnMap.containsKey(tableName.toLowerCase());
}

public boolean columnExistsInTable(String tableName, String
columnName) {
 Set<String> columns =
tableColumnMap.get(tableName.toLowerCase());
 return columns != null &&
columns.contains(columnName.toLowerCase());
}

```

```

 }

 public String getColumnType(String tableName, String columnName)
 {
 Map<String, String> columns =
 columnTypeMap.get(tableName.toLowerCase());
 return columns != null ?
 columns.get(columnName.toLowerCase()) : null;
 }

 public Set<String> getAllTables() {
 return tableColumnMap.keySet();
 }

 public Set<String> getColumnsForTable(String tableName) {
 return tableColumnMap.getDefault(tableName.toLowerCase(),
 Collections.emptySet());
 }
}

```

### Step 3: Create the SQL File Parser

```

java
@Component
public class SqlFileParser {

 public List<SqlQuery> parseSqlFile(String content) {
 List<SqlQuery> queries = new ArrayList<>();

 String[] lines = content.split("\n");
 StringBuilder currentQuery = new StringBuilder();
 int startLine = 1;

 for (int i = 0; i < lines.length; i++) {
 String line = lines[i].trim();

 // Skip comments and empty lines
 if (line.isEmpty() || line.startsWith("--") ||
 line.startsWith("#")) {
 if (currentQuery.length() == 0) {
 startLine = i + 2; // Next line will be start
 line
 }
 continue;
 }
 }
 }
}

```

```

 currentQuery.append(line).append(" ");

 // Check if line ends with semicolon (end of query)
 if (line.endsWith(";")) {
 String query = currentQuery.toString().trim();
 queries.add(new SqlQuery(query, startLine));

 // Reset for next query
 currentQuery = new StringBuilder();
 startLine = i + 2; // Next line will be start line
 }
 }

 // Handle last query if it doesn't end with a semicolon
 if (currentQuery.length() > 0) {
 String query = currentQuery.toString().trim();
 if (!query.isEmpty()) {
 queries.add(new SqlQuery(query, startLine));
 }
 }

 return queries;
}
}

```

#### Step 4: Create the Syntax Validator with ANTLR Integration

```

java
@Component
public class AntlrSyntaxValidator {

 public ValidationResult validateSyntax(SqlQuery query) {
 ValidationResult result = new ValidationResult();

 try {
 // This is where you'd integrate your existing ANTLR
 parser
 // The implementation depends on your ANTLR grammar and
 parser

 // Example (replace with your actual implementation):
 /*
 CharStream input =
 CharStreams.fromString(query.getSql());

```

```

 YourSqlLexer lexer = new YourSqlLexer(input);
 CommonTokenStream tokens = new CommonTokenStream(lexer);
 YourSqlParser parser = new YourSqlParser(tokens);

 // Add an error listener to capture syntax errors
 SyntaxErrorListener errorListener = new
SyntaxErrorListener();
 parser.removeErrorListeners();
 parser.addErrorListener(errorListener);

 // Start parsing at your root rule
 parser.yourRootRule();

 // Check if any syntax errors were found
 if (errorListener.hasErrors()) {
 for (SyntaxError error : errorListener.getErrors()) {
 result.addError(new ValidationError(
 ErrorType.SYNTAX,
 error.getMessage(),
 query.getLineNumber() + error.getLine() - 1,
 error.getCharPositionInLine()
));
 }
 }
 */

 // If no errors, the syntax is valid

 } catch (Exception e) {
 // Handle any exceptions from the parser
 result.addError(new ValidationError(
 ValidationError.ErrorType.SYNTAX,
 "Syntax error: " + e.getMessage(),
 query.getLineNumber(),
 0
));
 }

 return result;
}

// Custom error listener to capture ANTLR syntax errors
/*
private static class SyntaxErrorListener extends
BaseErrorListener {

```

```

 private List<SyntaxError> errors = new ArrayList<>();

 @Override
 public void syntaxError(Recognizer<?, ?> recognizer, Object
offendingSymbol,
 int line, int charPositionInLine,
 String msg, RecognitionException e) {
 errors.add(new SyntaxError(line, charPositionInLine,
msg));
 }

 public boolean hasErrors() {
 return !errors.isEmpty();
 }

 public List<SyntaxError> getErrors() {
 return errors;
 }
 }

 private static class SyntaxError {
 private int line;
 private int charPositionInLine;
 private String message;

 // Constructor, getters
 }
 */
}

```

## Step 5: Create the Schema Validator

```

java
@Component
public class SchemaValidator {
 private final SchemaMetadataService schemaService;
 private final JdbcTemplate jdbcTemplate;

 @Autowired
 public SchemaValidator(SchemaMetadataService schemaService,
JdbcTemplate jdbcTemplate) {
 this.schemaService = schemaService;
 this.jdbcTemplate = jdbcTemplate;
 }
}

```

```

 public ValidationResult validateSchema(SqlQuery query) {
 ValidationResult result = new ValidationResult();

 try {
 // Extract table references using DatabaseMetaData
 Set<String> referencedTables =
 extractTableReferences(query.getSql());

 // Validate tables exist
 for (String table : referencedTables) {
 if (!schemaService.tableExists(table)) {
 result.addError(new ValidationError(
 ValidationError.ErrorType.UNKNOWN_TABLE,
 "Unknown table: " + table,
 query.getLineNumber(),
 0
));
 }
 }

 // Extract and validate column references
 Map<String, Set<String>> columnsByTable =
 extractColumnReferences(query.getSql());

 for (Map.Entry<String, Set<String>> entry :
 columnsByTable.entrySet()) {
 String tableName = entry.getKey();

 // Skip if table doesn't exist (already reported)
 if (!schemaService.tableExists(tableName)) {
 continue;
 }

 // Check columns exist in table
 for (String columnName : entry.getValue()) {
 if (!schemaService.columnExistsInTable(tableName,
 columnName)) {
 result.addError(new ValidationError(
 ValidationError.ErrorType.UNKNOWN_COLUMN,
 "Unknown column '" + columnName + "' in
 table '" + tableName + "'",
 query.getLineNumber(),
 0
));
 }
 }
 }
 }
 }

```

```

 }
}

 // Validate data types in INSERT/UPDATE statements
 if (isDataModifyingQuery(query.getSql())) {
 validateDataTypes(query, result);
 }

 } catch (Exception e) {
 result.addError(new ValidationError(
 ValidationError.ErrorType.OTHER,
 "Schema validation error: " + e.getMessage(),
 query.getLineNumber(),
 0
));
 }

 return result;
}

private Set<String> extractTableReferences(String sql) {
 Set<String> tables = new HashSet<>();

 // Using a simplified regex approach - this can be enhanced
 // for better accuracy or replaced with a proper SQL parser
 Pattern tablePattern = Pattern.compile(
 "\\b(FROM|JOIN|UPDATE|INTO)\\s+(['\"]?([\\w_]+)['\""]?)",
 Pattern.CASE_INSENSITIVE
);

 Matcher matcher = tablePattern.matcher(sql);
 while (matcher.find()) {
 tables.add(matcher.group(3).toLowerCase());
 }

 return tables;
}

private Map<String, Set<String>> extractColumnReferences(String
sql) {
 Map<String, Set<String>> columnsByTable = new HashMap<>();

 // Extract qualified column references (table.column)
 Pattern qualifiedColumnPattern = Pattern.compile(

```



```

 "([`'\"?](\\w_+)[`'\"?])\\.([`'\"?](\\w_+)[`'\"?])",
 Pattern.CASE_INSENSITIVE
);

 Matcher matcher = qualifiedColumnPattern.matcher(sql);
 while (matcher.find()) {
 String tableName = matcher.group(2).toLowerCase();
 String columnName = matcher.group(4).toLowerCase();

 columnsByTable
 .computeIfAbsent(tableName, k -> new HashSet<>())
 .add(columnName);
 }

 // For unqualified columns, we'd need more complex parsing
 // This is simplified and would miss many cases

 return columnsByTable;
}

private boolean isDataModifyingQuery(String sql) {
 String upperSql = sql.trim().toUpperCase();
 return upperSql.startsWith("INSERT") ||
 upperSql.startsWith("UPDATE") ||
 upperSql.startsWith("DELETE");
}

private void validateDataTypes(SqlQuery query, ValidationResult
result) {
 // This would require more complex SQL parsing to extract
 // values and their target columns, then compare types

 // For a production system, you might want to use
 // a proper SQL parser or consider safe execution
 // in a transaction with rollback
}
}

```

## Step 6: Create the Main Query Validator Service

```

java
@Service
public class QueryValidatorService {
 private final AntlrSyntaxValidator syntaxValidator;
 private final SchemaValidator schemaValidator;
}

```

```

private final SqlFileParser fileParser;
private final JdbcTemplate jdbcTemplate;

@Autowired
public QueryValidatorService(
 AntlrSyntaxValidator syntaxValidator,
 SchemaValidator schemaValidator,
 SqlFileParser fileParser,
 JdbcTemplate jdbcTemplate) {
 this.syntaxValidator = syntaxValidator;
 this.schemaValidator = schemaValidator;
 this.fileParser = fileParser;
 this.jdbcTemplate = jdbcTemplate;
}

public ValidationResult validateSqlFile(String fileContent) {
 ValidationResult overallResult = new ValidationResult();

 // Parse the SQL file into individual queries
 List<SqlQuery> queries =
fileParser.parseSqlFile(fileContent);

 // Validate each query
 for (SqlQuery query : queries) {
 // Step 1: Syntax validation
 ValidationResult syntaxResult =
syntaxValidator.validateSyntax(query);

 // Add syntax errors to overall result
 for (ValidationError error : syntaxResult.getErrors()) {
 overallResult.addError(error);
 }

 // Skip schema validation if syntax is invalid
 if (!syntaxResult.isValid()) {
 continue;
 }

 // Step 2: Schema validation
 ValidationResult schemaResult =
schemaValidator.validateSchema(query);

 // Add schema errors to overall result
 for (ValidationError error : schemaResult.getErrors()) {
 overallResult.addError(error);
 }
 }
}

```

```

 }

 // Step 3: Execute Plan Analysis for SELECT queries
 if
(query.getSql().trim().toUpperCase().startsWith("SELECT")) {
 analyzeExecutionPlan(query, overallResult);
 }

}

return overallResult;
}

private void analyzeExecutionPlan(SqlQuery query,
ValidationResult result) {
 try {
 List<Map<String, Object>> planRows =
jdbcTemplate.queryForList("EXPLAIN " + query.getSql());

 // Analyze plan for potential issues
 for (Map<String, Object> row : planRows) {
 if ("ALL".equals(row.get("type"))) {
 result.addWarning("Full table scan detected on
table '" +
row.get("table") + "' at line " +
query.getLineNumber());
 }

 if (row.get("rows") != null) {
 long rowCount = ((Number)
row.get("rows")).longValue();
 if (rowCount > 1000) {
 result.addWarning("Query at line " +
query.getLineNumber() +
" might process a large number of rows ("
+ rowCount + ")");
 }
 }
 }
 } catch (Exception e) {
 // Don't mark as error, just add warning
 result.addWarning("Could not analyze execution plan for
query at line " +
query.getLineNumber() + ": " + e.getMessage());
 }
}
}

```

```
}
```

## Step 7: Create REST Controller for API Access

```
java
@RestController
@RequestMapping("/api/validator")
public class QueryValidatorController {

 private final QueryValidatorService validatorService;

 @Autowired
 public QueryValidatorController(QueryValidatorService
validatorService) {
 this.validatorService = validatorService;
 }

 @PostMapping("/validate")
 public ResponseEntity<ValidationResult> validateSqlFile(
 @RequestParam("file") MultipartFile file) {

 try {
 String fileContent = new String(file.getBytes(),
StandardCharsets.UTF_8);
 ValidationResult result =
validatorService.validateSqlFile(fileContent);
 return ResponseEntity.ok(result);
 } catch (IOException e) {
 ValidationResult result = new ValidationResult();
 result.addError(new ValidationError(
 ValidationError.ErrorType.OTHER,
 "Failed to read uploaded file: " + e.getMessage(),
 0,
 0
));
 return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body(result);
 }
 }
}
```

## Phase 3: Enhancing the Validator

### Step 1: Improve Object Reference Extraction

For a more robust solution, consider extending the schema validator with better SQL parsing:

```
java
@Component
public class EnhancedSchemaValidator extends SchemaValidator {

 @Override
 protected Map<String, Set<String>> extractColumnReferences(String
sql) {
 Map<String, Set<String>> result = new HashMap<>();

 // First, get all tables in the query
 Set<String> tables = extractTableReferences(sql);

 try (Connection conn = dataSource.getConnection()) {
 // For each query type, extract expected columns
 if (isSelectQuery(sql)) {
 extractSelectQueryColumns(sql, tables, result, conn);
 } else if (isInsertQuery(sql)) {
 extractInsertQueryColumns(sql, result, conn);
 } else if (isUpdateQuery(sql)) {
 extractUpdateQueryColumns(sql, result, conn);
 }
 }

 } catch (SQLException e) {
 // Handle exception
 }

 return result;
 }

 private void extractSelectQueryColumns(String sql, Set<String>
tables,
 Map<String, Set<String>>
result,
 Connection conn) throws
SQLException {
 // Use JDBC metadata to help with column resolution
 PreparedStatement stmt = null;
 try {
 stmt = conn.prepareStatement(sql);
 ResultSetMetaData metaData = stmt.getMetaData();

 for (int i = 1; i <= metaData.getColumnCount(); i++) {
 String tableName = metaData.getTableName(i);
```

```

 String columnName = metaData.getColumnName(i);

 if (tableName != null && !tableName.isEmpty()) {
 result.computeIfAbsent(tableName.toLowerCase(), k
-> new HashSet<>())
 .add(columnName.toLowerCase());
 } else {
 // Handle unqualified columns by checking all
tables
 for (String table : tables) {
 if (schemaService.columnExistsInTable(table,
columnName)) {
result.computeIfAbsent(table.toLowerCase(), k -> new HashSet<>())
 .add(columnName.toLowerCase());
 }
 }
 }
 } finally {
 if (stmt != null) {
 stmt.close();
 }
 }
}

// Additional methods for other query types...
}

```

## Step 2: Add Safe Execution Validation

For the most thorough validation, especially for complex queries:

```

java
@Component
public class SafeExecutionValidator {

 private final DataSource dataSource;

 @Autowired
 public SafeExecutionValidator(DataSource dataSource) {
 this.dataSource = dataSource;
 }
}

```

```

 public ValidationResult validateWithSafeExecution(SqlQuery query)
 {
 ValidationResult result = new ValidationResult();

 Connection conn = null;
 try {
 conn = dataSource.getConnection();

 // Disable auto-commit for transaction
 conn.setAutoCommit(false);

 // Set transaction to read-only if possible
 if (!isDataModifyingQuery(query.getSql())) {
 conn.setReadOnly(true);
 }

 // Prepare the statement
 try (PreparedStatement stmt =
 conn.prepareStatement(query.getSql())) {
 // For SELECT queries, we can limit rows
 if
 (query.getSql().trim().toUpperCase().startsWith("SELECT")) {
 stmt.setMaxRows(1);
 stmt.setFetchSize(1);

 // Execute and get metadata
 try (ResultSet rs = stmt.executeQuery()) {
 // Just getting metadata is enough for
 validation
 ResultSetMetaData metaData =
 rs.getMetaData();
 }
 } else {
 // For data-modifying queries, execute but roll
 back
 stmt.execute();
 }

 // Always roll back to avoid any data changes
 conn.rollback();

 } catch (SQLException e) {
 result.addError(new ValidationError(
 ValidationError.ErrorType.OTHER,

```

```

 "Query execution error: " + e.getMessage(),
 query.getLineNumber(),
 0
));
 } finally {
 if (conn != null) {
 try {
 conn.rollback();
 conn.close();
 } catch (SQLException e) {
 // Log cleanup error
 }
 }
 }

 return result;
}

private boolean isDataModifyingQuery(String sql) {
 String upperSql = sql.trim().toUpperCase();
 return upperSql.startsWith("INSERT") ||
 upperSql.startsWith("UPDATE") ||
 upperSql.startsWith("DELETE");
}
}

```

## Phase 4: Frontend Integration

Create a simple frontend for uploading SQL files:

```

html
<!DOCTYPE html>
<html>
<head>
 <title>SQL Validator</title>
 <style>
 body { font-family: Arial, sans-serif; max-width: 800px;
margin: 0 auto; padding: 20px; }
 .error { color: red; }
 .warning { color: orange; }
 .success { color: green; }
 pre { background-color: #f5f5f5; padding: 10px;
border-radius: 5px; }
 </style>

```



```
</head>
<body>
 <h1>MariaDB SQL Validator</h1>

 <form id="upload-form">
 <input type="file" id="sql-file" accept=".sql">
 <button type="submit">Validate</button>
 </form>

 <div id="results" style="margin-top: 20px;"></div>

 <script>

document.getElementById('upload-form').addEventListener('submit',
async function(e) {
 e.preventDefault();

 const fileInput = document.getElementById('sql-file');
 if (!fileInput.files.length) {
 alert('Please select an SQL file to validate');
 return;
 }

 const formData = new FormData();
 formData.append('file', fileInput.files[0]);

 try {
 const response = await
fetch('/api/validator/validate', {
 method: 'POST',
 body: formData
});

 const result = await response.json();
 displayResults(result);
 } catch (error) {
 alert('Error during validation: ' + error.message);
 }
});

function displayResults(result) {
 const resultsDiv = document.getElementById('results');

 if (result.valid) {
```

```

 resultsDiv.innerHTML = '<div
class="success"><h2>Validation Successful!</h2></div>';

 if (result.warnings.length > 0) {
 resultsDiv.innerHTML += '<h3>Warnings:</h3>';
 result.warnings.forEach(warning => {
 resultsDiv.innerHTML += '<li
class="warning">${warning}`;
 });
 resultsDiv.innerHTML += '';
 } else {
 resultsDiv.innerHTML = '<div
class="error"><h2>Validation Failed!</h2></div>';

 resultsDiv.innerHTML += '<h3>Errors:</h3>';
 result.errors.forEach(error => {
 let location = '';
 if (error.lineNumber > 0) {
 location = ` (Line: ${error.lineNumber})`;
 }

 resultsDiv.innerHTML += '<li
class="error">${error.type}: ${error.message}${location}`;
 });
 resultsDiv.innerHTML += '';

 if (result.warnings.length > 0) {
 resultsDiv.innerHTML += '<h3>Warnings:</h3>';
 result.warnings.forEach(warning => {
 resultsDiv.innerHTML += '<li
class="warning">${warning}`;
 });
 resultsDiv.innerHTML += '';
 }
 }
 }
</script>
</body>
</html>

```

## Integration Testing

Create tests to verify your validator works correctly:

```

java
@SpringBootTest
public class QueryValidatorIntegrationTest {

 @Autowired
 private QueryValidatorService validatorService;

 @Test
 public void testValidQuery() {
 String sql = "SELECT * FROM valid_table;";
 ValidationResult result =
validatorService.validateSqlFile(sql);
 assertTrue(result.isValid());
 }

 @Test
 public void testSyntaxError() {
 String sql = "SELCET * FROM table;"; // Typo in SELECT
 ValidationResult result =
validatorService.validateSqlFile(sql);
 assertFalse(result.isValid());
 assertTrue(result.getErrors().stream()
 .anyMatch(e -> e.getType() ==
ValidationError.ErrorType.SYNTAX));
 }

 @Test
 public void testUnknownTable() {
 String sql = "SELECT * FROM nonexistent_table;";
 ValidationResult result =
validatorService.validateSqlFile(sql);
 assertFalse(result.isValid());
 assertTrue(result.getErrors().stream()
 .anyMatch(e -> e.getType() ==
ValidationError.ErrorType.UNKNOWN_TABLE));
 }

 @Test
 public void testUnknownColumn() {
 String sql = "SELECT nonexistent_column FROM valid_table;";
 ValidationResult result =
validatorService.validateSqlFile(sql);
 assertFalse(result.isValid());
 assertTrue(result.getErrors().stream()

```

```
 .anyMatch(e -> e.getType() ==
ValidationError.ErrorType.UNKNOWN_COLUMN));
 }
}
```

## Final Implementation Checklist

1. ANTLR Integration
  - Ensure your existing ANTLR parser is properly integrated with the AntlrSyntaxValidator
2. Schema Validation Enhancement
  - Consider enhancing table/column extraction for complex queries
3. Safe Execution Environment
  - Configure a test database with the same schema as production
  - Ensure proper transaction handling with rollback
4. Performance Optimization
  - Implement schema caching with appropriate refresh intervals
  - Add query result caching for repeated validation
5. Security Considerations
  - Use read-only database user for validation when possible
  - Sanitize error messages to avoid exposing sensitive information
6. Documentation
  - Document the API endpoints
  - Create usage examples for integration with CI/CD pipelines

This step-by-step guide should provide you with a solid foundation for building your MariaDB query validator. The hybrid approach leverages your existing ANTLR parser for syntax validation while using JDBC metadata for schema validation. The result is a comprehensive validation tool that can catch syntax errors, schema inconsistencies, and potential performance issues before queries are deployed to production.