keystoneDB

Group Name: ChadGPT
Mayash Nayak 22CS30064
Sumit Kumar 22CS30056
Aviral Singh 22CS30015

June 24, 2025

Project Link: keystoneDB: https://github.com/SumitKumar-17/keystoneDB

Contents

1 Introduction

1.1 Project Overview

keystoneDB is an educational DBMS implementation designed to provide hands-on experience with database internals. The system implements a subset of SQL functionality with a focus on understanding the core components of a database management system. The project demonstrates practical implementation of concepts including:

- SQL lexical analysis and parsing
- Translation of SQL statements to an intermediate representation
- Execution of queries using a custom engine
- Persistent storage using a key-value store (RocksDB)
- Data type implementation and constraint enforcement
- Expression evaluation using the visitor pattern

1.2 Educational Goals

The primary objective of keystoneDB is educational, focusing on:

- Understanding compiler design principles in the context of SQL
- Gaining practical experience with expression evaluation using the visitor pattern
- Learning integration techniques for third-party libraries
- Practicing unit testing with GoogleTest
- Applying modern C++ programming practices

2 System Architecture

2.1 High-Level Architecture

The keystoneDB system follows a layered architecture typical of database management systems:

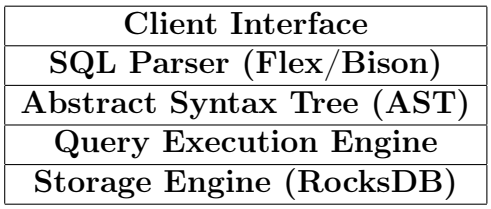

Figure 1: High-level architecture of keystoneDB

Each layer has a specific responsibility:

- **Client Interface**: Provides a command-line interface with history and line-editing capabilities
- **SQL Parser**: Converts SQL text into tokens and builds an abstract syntax tree
- **Abstract Syntax Tree**: Represents the structure of SQL statements
- **Query Execution Engine**: Executes the statements represented by the AST
- **Storage Engine**: Handles persistent storage using RocksDB

2.2 Visual Architecture Overview

keystoneDB's architecture can be visualized through the following component diagrams that illustrate the system's key modules and their interactions.

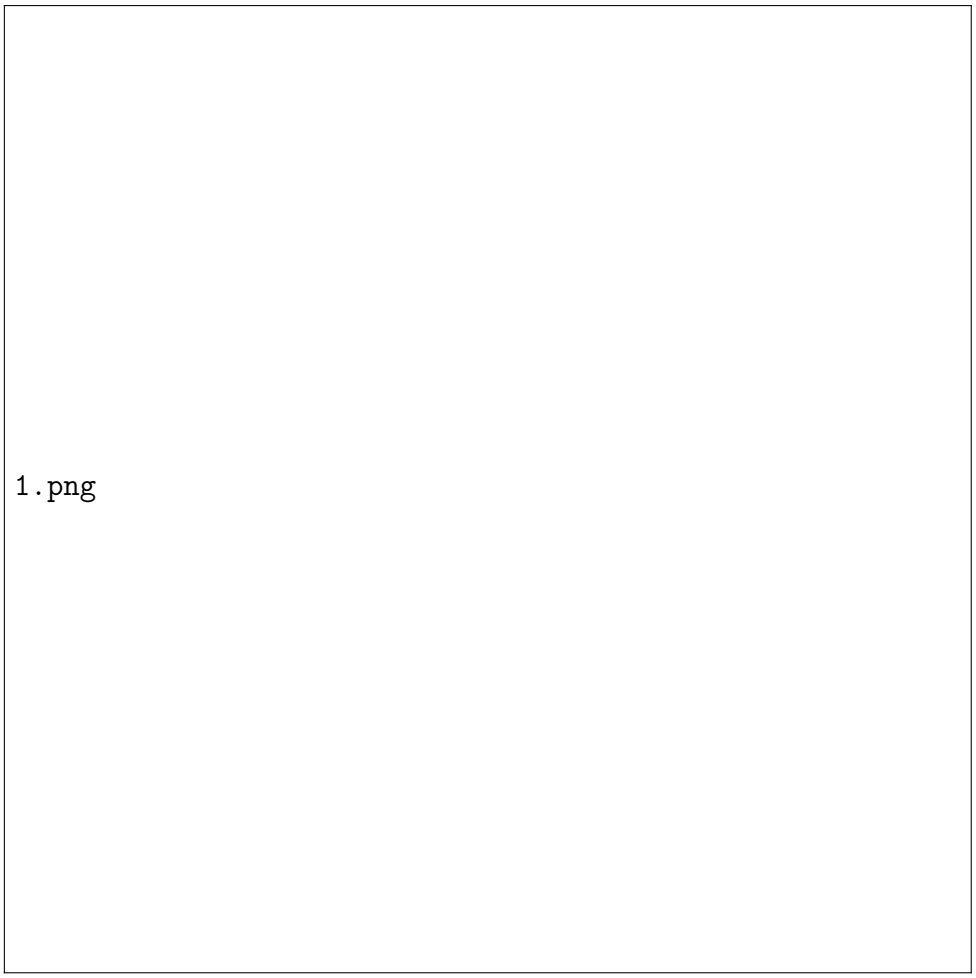

Figure 2: Component interaction overview

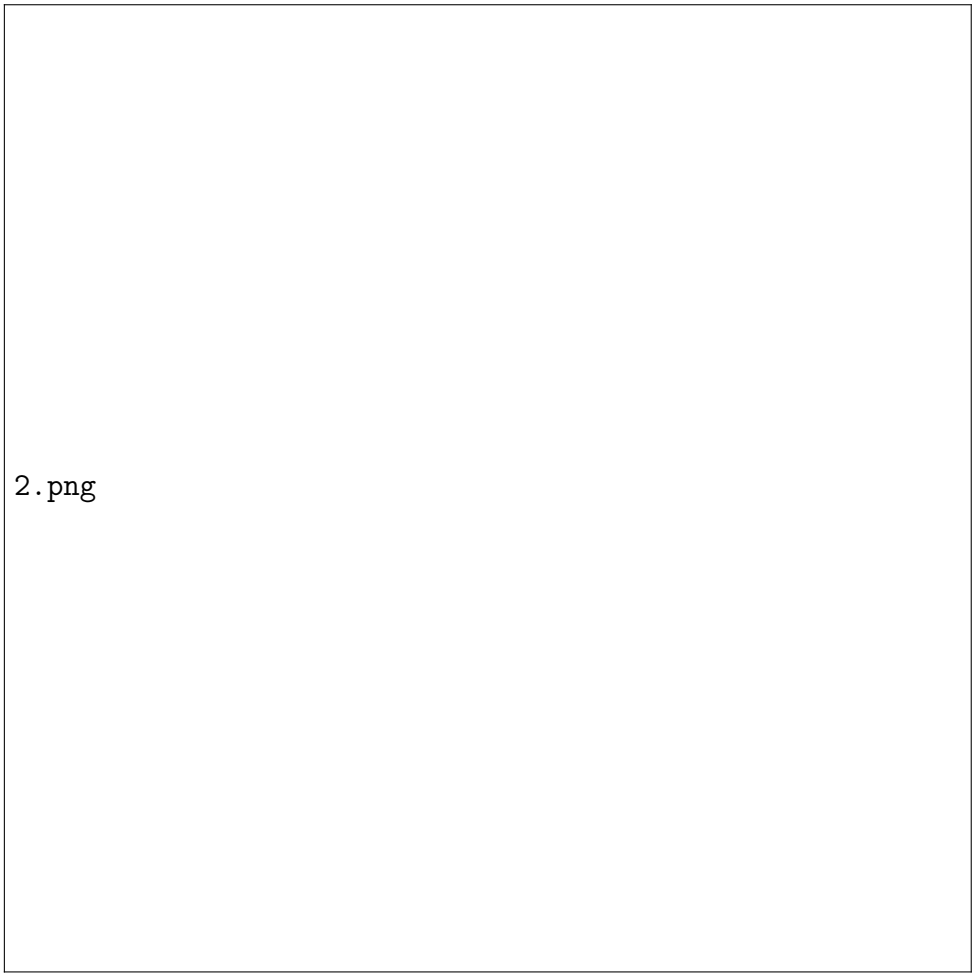

Figure 3: Parser and AST generation pipeline

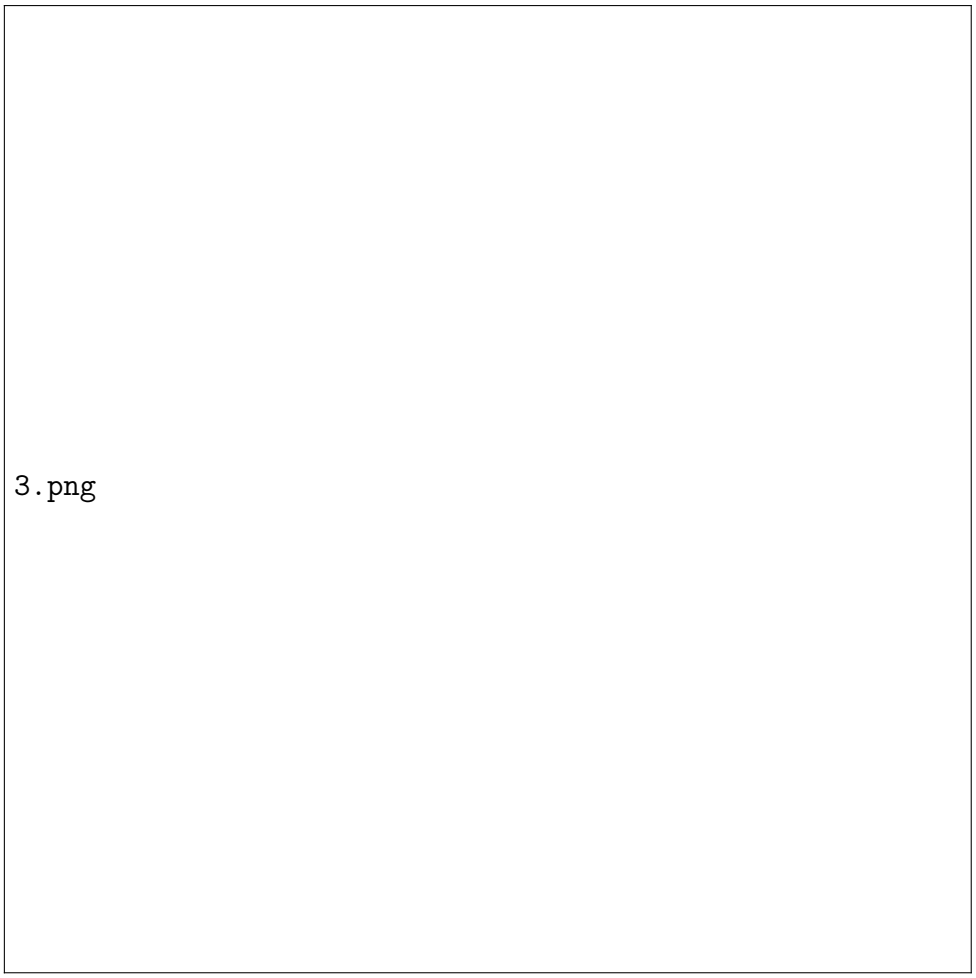

Figure 4: Query execution workflow

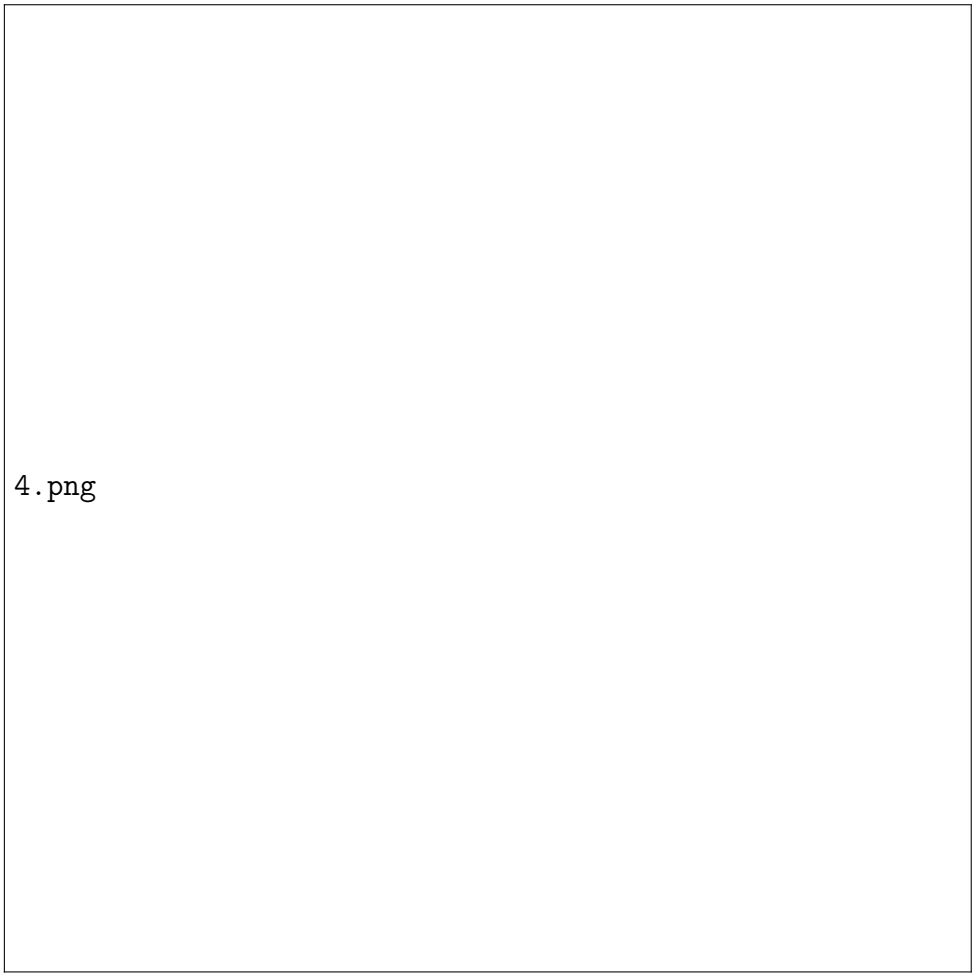

Figure 5: Storage engine integration

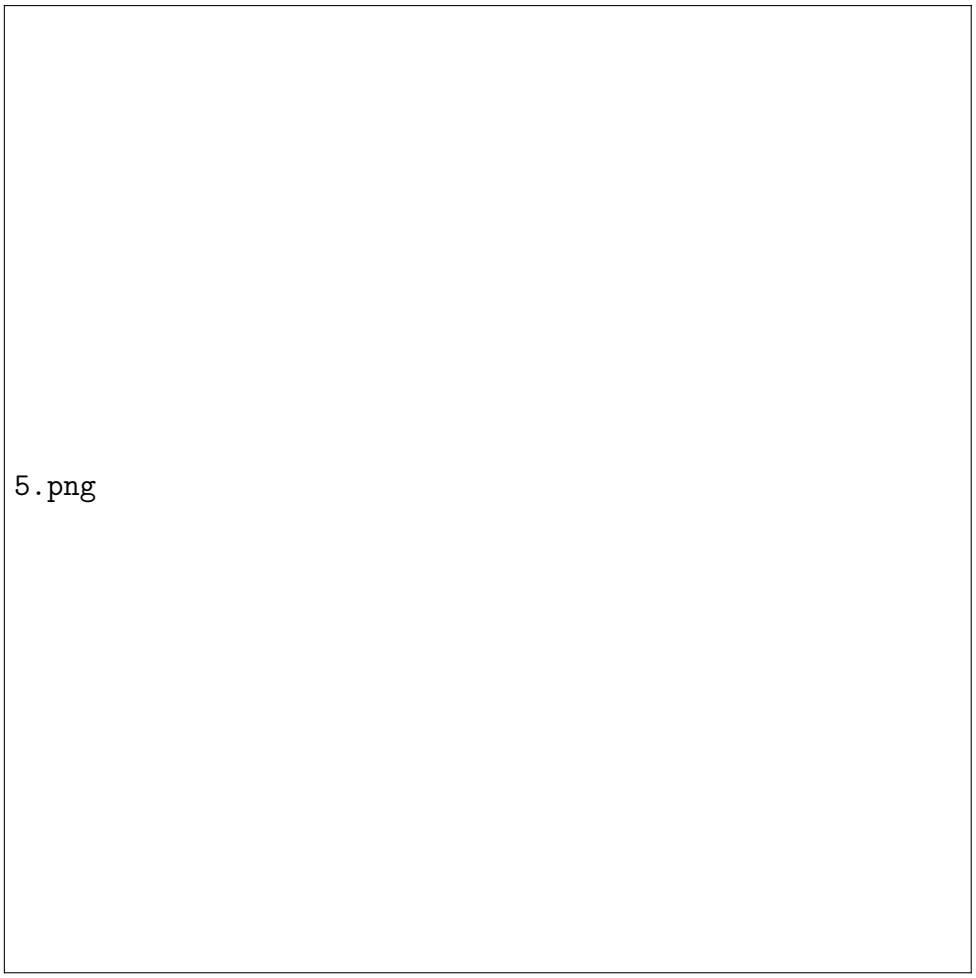

Figure 6: Expression evaluation system

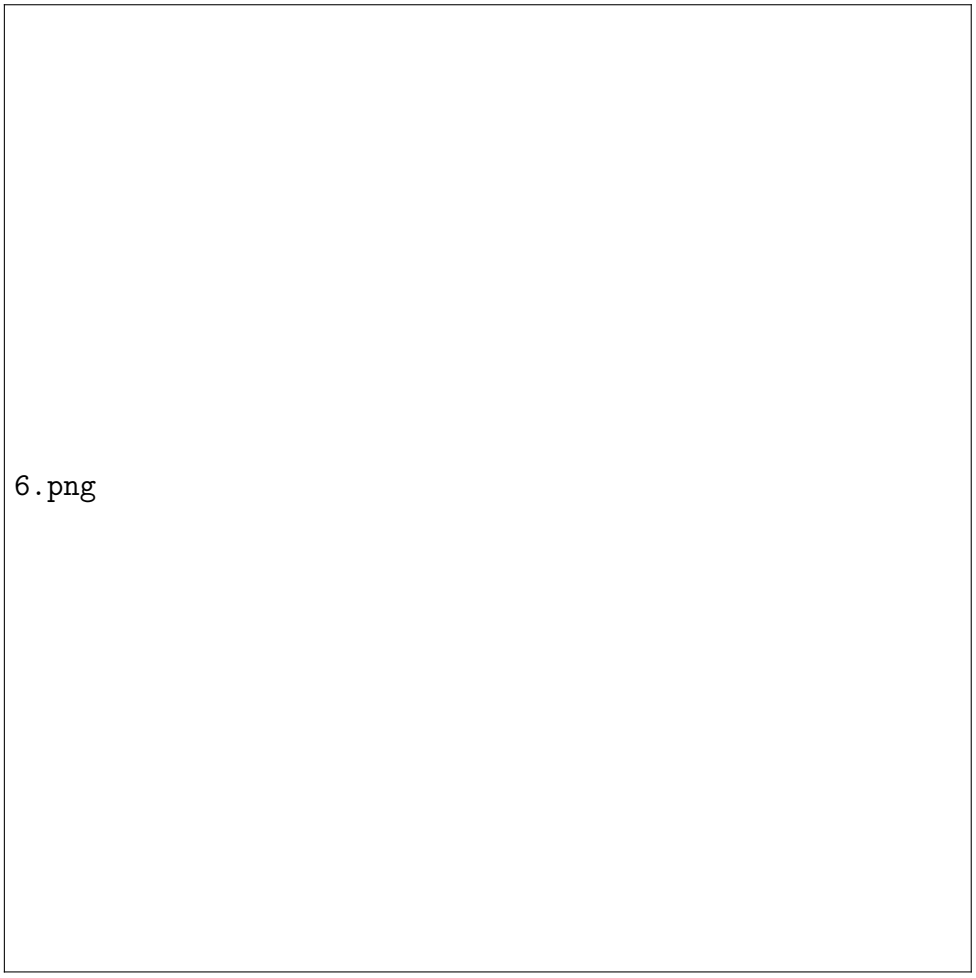

Figure 7: Data type hierarchy

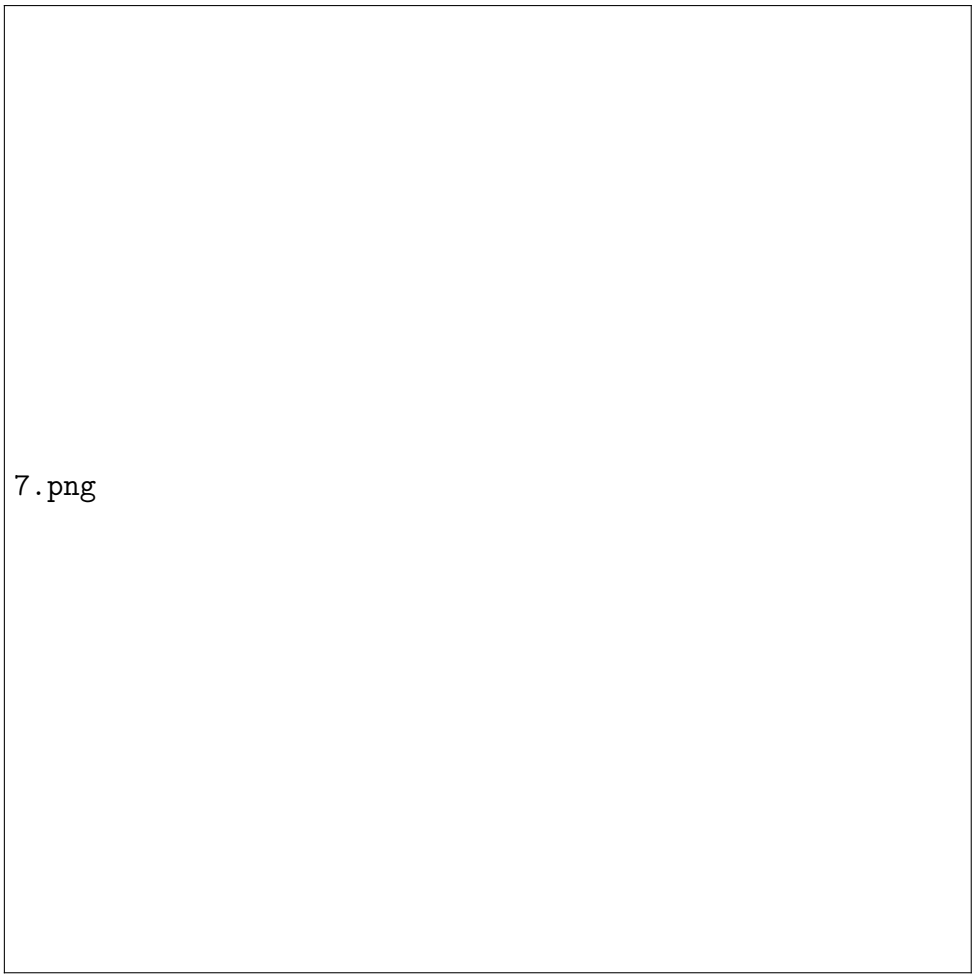

Figure 8: Constraint enforcement mechanism

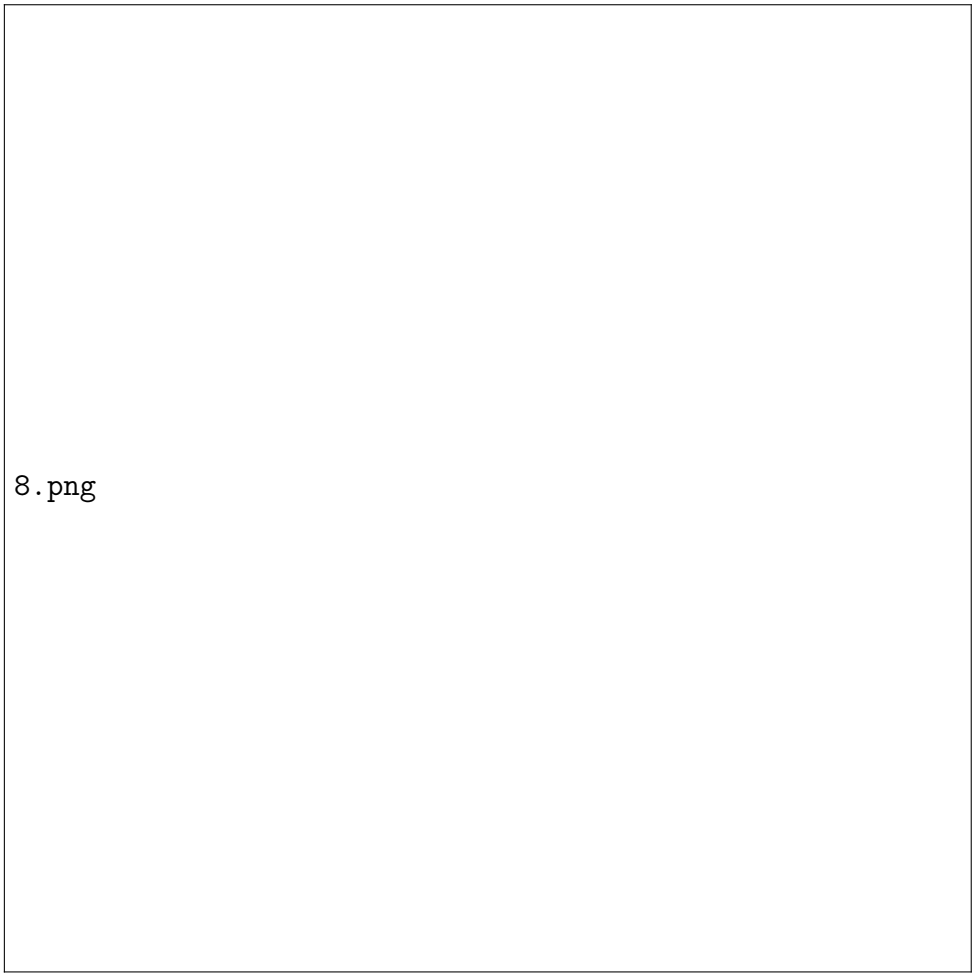

Figure 9: Transaction management

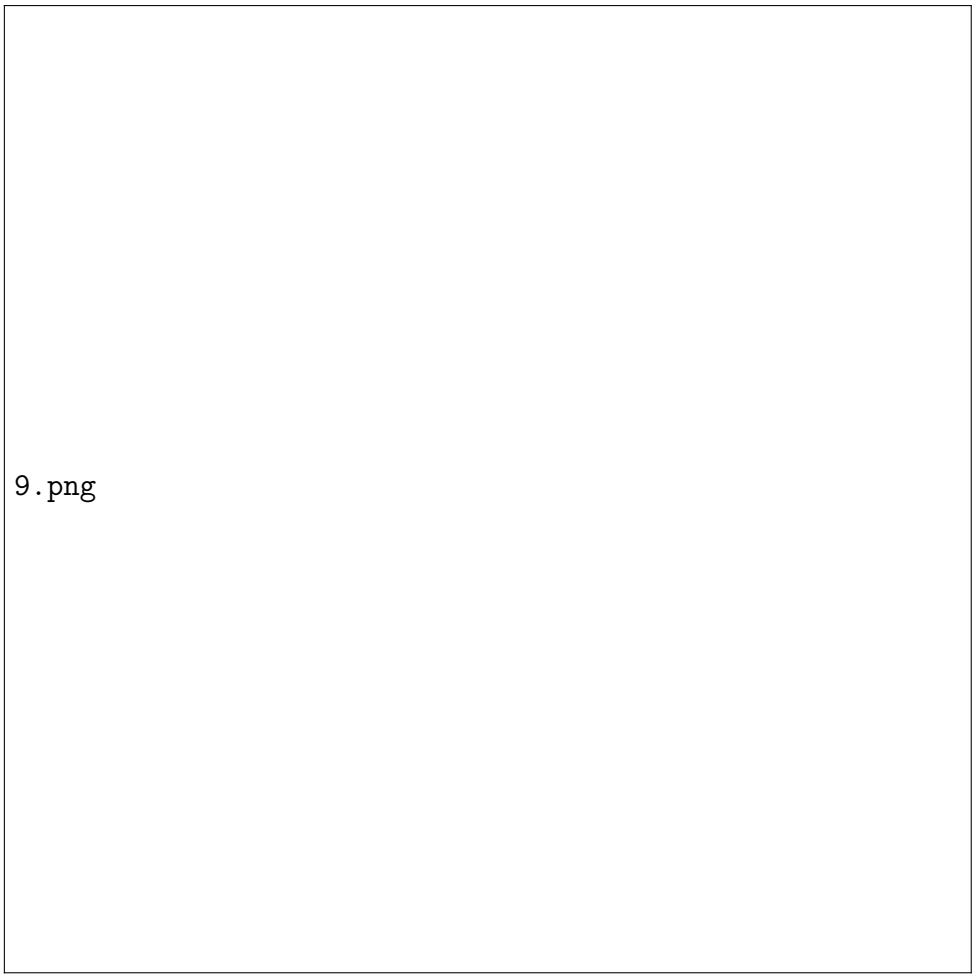

Figure 10: Error handling subsystem

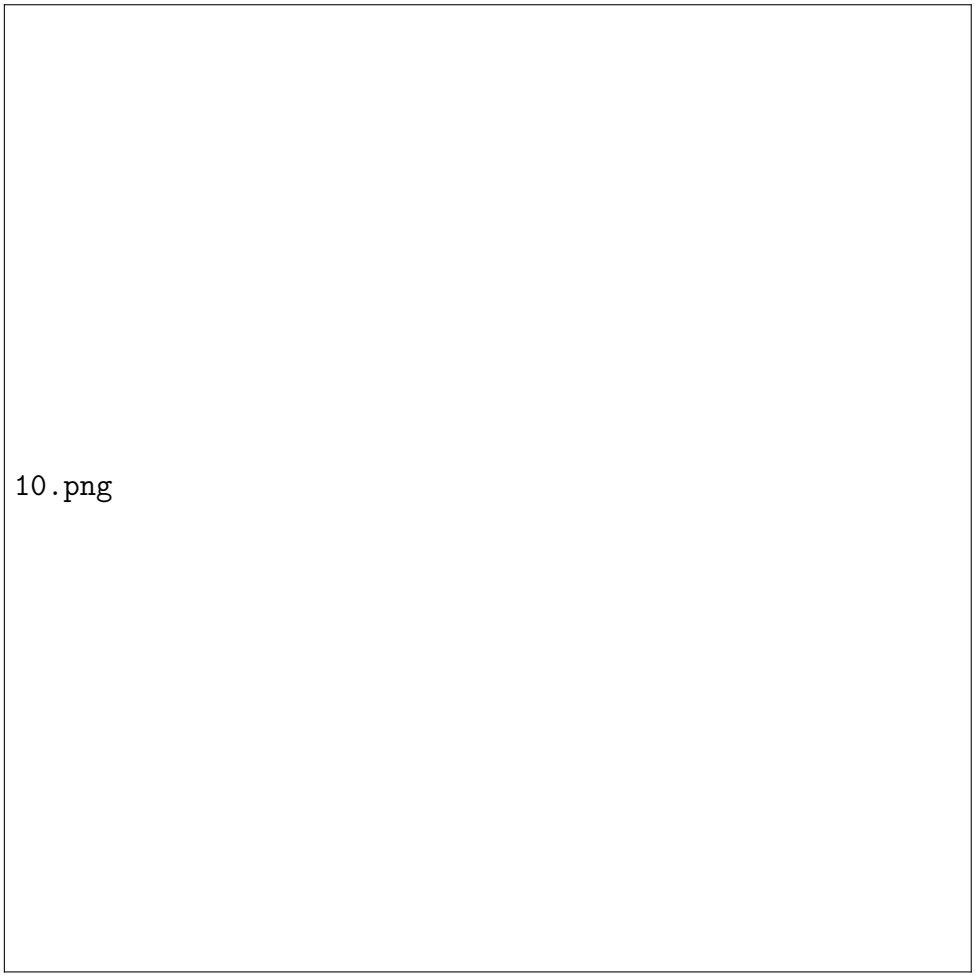

Figure 11: Testing infrastructure

2.3 Key Architectural Components

The architecture diagrams above illustrate keystoneDB's modular design with clear separation of concerns:

- **Parser Subsystem**: Converts SQL text into a structured AST representation
- **Execution Engine**: Translates parsed statements into operations on the storage layer
- **Storage Manager**: Provides persistence through RocksDB integration
- **Type System**: Enforces data type constraints and conversions
- **Expression Evaluator**: Processes complex query conditions using the visitor pattern

This layered approach facilitates maintenance, testing, and extensibility of the system while providing clear educational examples of database internals.

3 Parser Implementation

3.1 Lexical Analysis

The lexical analyzer (scanner) is implemented using Flex in `sql.l`. The scanner demonstrates several advanced techniques:

- **Reentrant design**: Uses `%option reentrant` for thread safety
- **Case-insensitive token matching**: For SQL keywords
- **Start conditions**: For handling complex tokens like strings and comments
- **Thread-local string buffers**: For safely building string literals

```
%option reentrant
%%
(?i:SELECT)    { return SELECT; }
(?i:FROM)      { return FROM; }
(?i:WHERE)     { return WHERE; }
```

Listing 1: Example of a lexer rule for SQL keywords

The lexer recognizes the full SQL token spectrum including keywords, operators, identifiers, and literals. It provides detailed error reporting, particularly for unterminated strings and unknown characters.

3.2 Syntax Analysis

The syntax analyzer (parser) is implemented using Bison in `sql.y`. The parser implements a comprehensive SQL grammar with:

- **Type-safe value passing**: Through a `%union` structure
- **Precise operator precedence**: Definitions ensuring expressions like `a + b * c` are parsed correctly
- **Memory management**: Via destructors to prevent leaks from string allocations
- **Error recovery**: Mechanisms for syntax mistakes

```
%union {
    int64_t ival;
    double fval;
    char *str;
    keystoneDB::SQLStmt *stmt;
    keystoneDB::ColumnDef *col_def;
    keystoneDB::TableDef *table_def;
    keystoneDB::DropStmt *drop_stmt;
    keystoneDB::Exp *exp;
    std::vector<keystoneDB::ColumnDef *> *col_list;
    std::vector<keystoneDB::Exp *> *exp_list;
    std::vector<keystoneDB::SelectCol *> *select_list;
    std::vector<keystoneDB::TableRef *> *table_list;
}

%token <str> IDENTIFIER STRING_LITERAL
%token <ival> INTEGER_LITERAL
%token <fval> FLOAT_LITERAL

%token CREATE DROP TABLE DATABASE SELECT
%token INSERT INTO VALUES UPDATE SET
%token DELETE FROM WHERE USE SHOW TABLES DATABASES
%token INT FLOAT CHAR NULL_T NOT IS BETWEEN AND OR
%token EXIT

%left OR
%left AND
%left '=' '<' '>' LE GE NE IS
%left '+' '-'
%left '*' '/' '%'
%left '^'
%right UMINUS
```

Listing 2: Example of union definition and token types

The parser handles a wide range of SQL statements including DDL operations (CREATE, DROP), DML operations (INSERT, UPDATE, DELETE), and queries (SELECT).

4 Abstract Syntax Tree Design

4.1 Statement Hierarchy

The project features a sophisticated object-oriented AST design with a hierarchy of SQL statement types:

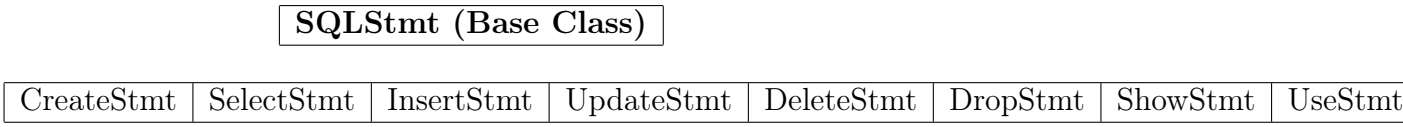

Figure 12: Inheritance hierarchy of SQL statements

All SQL statements derive from a common `SQLStmt` base class, with specialized implementations for different statement types:

- `CreateStmt` for table and database creation
- `SelectStmt` for data retrieval
- `InsertStmt` for adding records
- `UpdateStmt` for modifying data
- `DeleteStmt` for removing records
- `DropStmt` for schema modifications
- `ShowStmt` and `UseStmt` for utility operations

This hierarchy allows for polymorphic handling of different statement types throughout the execution pipeline.

4.2 Expression System

The expression handling system uses an elegant hierarchical design:

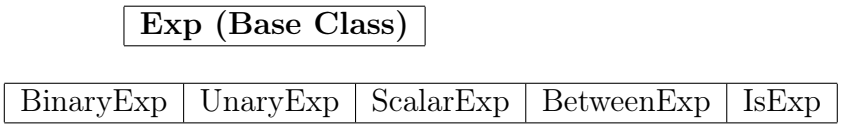

Figure 13: Expression hierarchy in keystoneDB

The expression system includes:

- `Exp` base class providing common functionality
- Specialized classes for different expression types
- Visitor pattern through `AbstractExpProcessor` to separate expression structure from operations
- Support for complex conditions including between expressions and logical operations

This design facilitates not just parsing but later stages like optimization and execution as well.

5 Storage Engine

5.1 RocksDB Integration

keystoneDB uses RocksDB as its persistent storage engine. RocksDB is an LSM-tree-based key-value store that provides high performance for write-heavy workloads.

```
rocksdb::DB *db;
rocksdb::Options options;
options.create_if_missing = true;
rocksdb::Status status = rocksdb::DB::Open(options, "/path/to/db", &db);
assert(status.ok());
```

Listing 3: Opening a RocksDB database

5.2 Data Encoding Strategy

The project uses a sophisticated encoding approach for mapping relational data to key-value pairs:

- **Table mapping**: Each table receives unique identifiers
- **Primary key encoding**: Records are stored with keys that include table ID and primary key values
- **Secondary index encoding**: Indexes are maintained as separate key-value entries pointing to primary keys
- **Schema storage**: Table definitions and constraints are stored as special system entries

This encoding approach is similar to what's used in MyRocks, a MySQL storage engine based on RocksDB.

5.3 Transaction and Consistency Management

The system handles transactions and consistency through:

- **Synchronous writes**: For durability-critical operations
- **Snapshots**: For consistent read views during transactions
- **Column families**: For logical separation with atomic write capabilities

```
rocksdb::WriteOptions write_options;
write_options.sync = true;
db->Put(write_options, ...);
```

Listing 4: Using synchronous writes in RocksDB

6 Query Execution

6.1 Execution Pipeline

The execution pipeline consists of several stages:

1. **Lexical analysis**: Converting SQL text to tokens
2. **Parsing**: Building an AST representation
3. **Semantic validation**: Checking against schema information
4. **Optimization**: Selecting efficient execution paths
5. **Execution**: Translating operations to RocksDB calls
6. **Result processing**: Formatting and returning query results

6.2 Statement Executors

Each SQL statement type has a corresponding executor:

- `CreateExecutor`: Handles CREATE TABLE and CREATE DATABASE statements
- `SelectExecutor`: Processes SELECT queries with filtering and joins
- `InsertExecutor`: Manages data insertion operations
- `UpdateExecutor`: Handles data modification
- `DeleteExecutor`: Processes data deletion with conditions
- `DropExecutor`: Manages DROP TABLE and DROP DATABASE statements
- `ShowExecutor`: Displays metadata like table list
- `UseExecutor`: Changes the current database context

6.3 Expression Evaluation

Expression evaluation is handled using the visitor pattern:

- `BinaryEvaluator`: Evaluates binary operations (+, -, *, /, etc.)
- `UnaryEvaluator`: Handles unary operations (-x, NOT x, etc.)
- `ScalarEvaluator`: Processes scalar values (literals, column references)
- `BetweenEvaluator`: Handles BETWEEN expressions
- `ExpChecker`: Validates expressions against schema

This approach separates the traversal of expression trees from the operations performed on them.

7 Type System and Schema Management

7.1 Data Types

keystoneDB supports a set of basic SQL data types:

- `INT`: For integer values
- `FLOAT`: For floating-point numbers
- `CHAR(N)`: For fixed-length character strings

7.2 Constraints

The system implements basic constraints:

- `NOT NULL`: Ensures a column cannot have NULL values
- Primary key support with uniqueness enforcement

```
CREATE TABLE t5 (id INT NOT NULL);
INSERT INTO t5 VALUES (1),(2);
-- The following would cause an error:
-- INSERT INTO t5 VALUES (NULL);
```

Listing 5: Example of constraint usage

7.3 Schema Storage

Schema information is stored within RocksDB, allowing persistence across database restarts and providing a single source of truth for metadata.

8 Testing

8.1 Unit Testing Framework

keystoneDB uses GoogleTest for unit testing:

```
// From create_stmt_test.cpp
TEST(CreateStmtTest, CreateTable) {
    // Test setup
    // ...

    // Create a table
    auto stmt = new CreateStmt(/*...*/);

    // Verify results
    EXPECT_EQ(/*...*/);
}
```

Listing 6: Example of a unit test for CreateStmt

8.2 SQL Test Suite

The project includes SQL test files:

```
create database test;
use test;
create table t1 (x int,y char(20));
insert into t1 values (42,'hello');
select * from t1;
insert into t1 values (43,'hello');
insert into t1 values (42,NULL);
insert into t1 values (NULL,NULL);
insert into t1 values (99,'hello'),(100,'world');
insert into t1 (x) values (101);
insert into t1 (y) values ('y only');

-- insert test (error expected)
-- Char too long
insert into t1 (x,y) values (101,'looooooooooooooooooooooooooooooooooog'
   );
```

Listing 7: Excerpt from test.sql

These test files cover a range of SQL functionalities including:

- Basic DDL operations (CREATE, DROP)
- DML operations (INSERT, UPDATE, DELETE)
- Queries with filtering (SELECT ... WHERE)
- Error handling cases
- Expression evaluation
- Constraint enforcement

9 Features and Capabilities

9.1 Supported SQL Syntax

keystoneDB supports a substantial subset of SQL:

```
CREATE DATABASE example;
USE example;
CREATE TABLE user (id INT, score FLOAT);
SHOW TABLES;
INSERT INTO user (id INT NOT NULL) VALUES (1);
SELECT id FROM user WHERE id = 42;
UPDATE user SET id=1 WHERE id=42;
DELETE FROM user WHERE id=42;
SELECT * FROM user WHERE id=(1+2*2+(id=id)+id^id+id) AND id = id%2 AND
   id IS NOT NULL;
SELECT * FROM t1 WHERE id IS NOT NULL;
DROP TABLE user;
```

Listing 8: Examples of supported SQL statements

9.2 Expression Evaluation

The system supports complex expressions:

- Arithmetic operations: +, -, *, /, %, ^

  Comparisonoperations :=, <, >, <=, >=, <>

- Logical operations: AND, OR
- Special conditions: IS NULL, IS NOT NULL, BETWEEN
- Nested expressions with proper precedence

```
SELECT * FROM t2 WHERE ('a'<y AND x>2)=x-1;
SELECT * FROM t6 WHERE id BETWEEN 2 AND 4;
SELECT * FROM t6 WHERE 3 BETWEEN id AND 4;
```

Listing 9: Examples of complex expressions

9.3 Multi-Table Operations

keystoneDB supports joins between tables:

```
-- No ambiguity
SELECT t2.x, t3.x FROM t2, t3;

-- Infer the table that z belongs to
SELECT t2.x, z FROM t2, t4;
```

Listing 10: Examples of multi-table operations

The system handles column ambiguity and qualification appropriately.

10 Error Handling

keystoneDB includes comprehensive error handling:

- **Syntax errors**: Detected during parsing
- **Semantic errors**: Like ambiguous column names
- **Constraint violations**: Such as NULL in NOT NULL columns
- **Type mismatches**: For example, string in integer column

```
-- Column id can not be NULL
INSERT INTO t5 VALUES (NULL);

-- Char too long
INSERT INTO t1 (x,y) VALUES (101,'looooooooooooooooooooooooooooooooooog'
    );

-- Data type mismatch
INSERT INTO t1 (x,y) VALUES ('100','mismatch');

```

```
-- Too many arguments
INSERT INTO t1 (x) VALUES (1,'column mismatch',99);

-- Column count doesn't match value count
INSERT INTO t1 VALUES (1);

-- Unknown column name
INSERT INTO t1 (not_exist_name) VALUES (1,'not exist name');

-- Ambiguous column name
SELECT x, y FROM t2, t3;
```

Listing 11: Examples of error cases

11 Conclusion

keystoneDB demonstrates a sophisticated approach to building an educational database management system. The project provides valuable insights into database internals through practical implementation.

The architecture makes thoughtful trade-offs between complexity and capability, with particular attention to:

- Memory management
- Type safety
- Separation of concerns
- SQL standard compliance
- Error handling

This educational DBMS follows the tradition of modern database systems that adapt key-value stores to serve relational workloads, providing a foundation for understanding both traditional relational database concepts and modern storage approaches.

12 References

1. Key-Value Database, https://en.wikipedia.org/wiki/Key-value_database
2. Parsing, https://en.wikipedia.org/wiki/Parsing
3. RocksDB, https://github.com/facebook/rocksdb
4. SQLite Logic Test, https://sqlite.org/sqllogictest/doc/trunk/about.wiki
5. Hyrise SQL Parser, https://github.com/hyrise/sql-parser