CoreDB SQL Engine
large Full Developer Manual with Deep Execution Walkthrough

CoreDB Internal Documentation

November 5, 2025

Contents

1 System Overview

CoreDB's SQL engine processes queries through four integrated layers:

1. **Lexer (Tokenizer)** – Splits SQL statements into tokens.
2. **Parser** – Builds Abstract Syntax Trees (ASTs) from tokens.
3. **Executor** – Executes the AST nodes and produces results.
4. **Storage Layer** – Persists data and maintains indexes in JSON.

Flow Overview

```
SQL Query
|
SQLTokenizer._tokenize()   -> Tokens
|
SQLParser.parse()          -> AST (e.g., InsertStatement)
|
QueryExecutor.execute()    -> Executes plan
|
StorageManager or IndexedStorageManager -> JSON files
```

2 Lexical Analysis (SQLTokenizer)

The lexer converts SQL text into tokens using regular expressions and keyword mapping.

How It Works

1. Loops through the SQL input with regex from the PATTERNS list.
2. Matches tokens: keywords, identifiers, literals, operators.
3. Maps identifiers to known keywords via the KEYWORDS dictionary.
4. Skips whitespace and comments.
5. Produces a sequential list of Token objects.

Example: INSERT

```
INSERT INTO BOOKS (BookID, Title, Price) VALUES (1, 'Data Mining', 700);
```

Tokens:

```
[INSERT, INTO, IDENTIFIER(BOOKS), LEFT_PAREN, IDENTIFIER(BookID), COMMA,
IDENTIFIER(Title), COMMA, IDENTIFIER(Price), RIGHT_PAREN, VALUES, LEFT_PAREN,
NUMBER_LITERAL(1), COMMA, STRING_LITERAL('Data Mining'), COMMA, NUMBER_LITERAL(700), RIGHT
```

3 Parser Internals

The parser constructs the AST based on SQL syntax rules. Each SQL command type triggers a specific parsing function.

```
SQLParser.parse()
|-CREATE -> _parse_create_table()
|-INSERT -> _parse_insert()
|-SELECT -> _parse_select()
|-UPDATE -> _parse_update()
|-DELETE -> _parse_delete()
|-DROP   -> _parse_drop_table()
|_ ALTER  -> _parse_alter_table()
```

Each function consumes expected tokens in sequence, validating syntax and building AST dataclasses.

3.1 Example: INSERT Parser

```
INSERT INTO BOOKS (BookID, Title, Price) VALUES (1, 'Data Mining', 700);
```

Function Trace:

```
_parse_insert()
|-expect(INSERT), expect(INTO)
|-table_name = 'BOOKS'
|-parse column list: ['BookID', 'Title', 'Price']
|-expect(VALUES)
|-parse value list: [1, 'Data Mining', 700]
|_ return InsertStatement(table='BOOKS', columns=[...], values=[...])
```

3.2 Example: SELECT with JOIN

```
SELECT M.MemberName, B.Title
FROM MEMBERS M
INNER JOIN ISSUE_DETAILS I ON M.MemberID = I.MemberID
INNER JOIN BOOKS B ON I.BookID = B.BookID;
```

Parser Steps:

```
_parse_select()
|-parse columns via _parse_column_expression()
|-parse FROM clause -> base table MEMBERS M
|-detect JOIN -> _parse_join_clause()
|     |-INNER JOIN ISSUE_DETAILS I ON M.MemberID = I.MemberID
|     |_ INNER JOIN BOOKS B ON I.BookID = B.BookID
|-build SelectStatement(columns, joins, ...)
|_ return AST
```

4 Query Execution Internals

The executor interprets AST nodes and interacts with the storage engine.

4.1 General Execution Flow

```
execute(ast)
|-if CREATE -> _execute_create_table()
|-if INSERT -> _execute_insert()
|-if SELECT -> _execute_select()
|-if UPDATE -> _execute_update()
|-if DELETE -> _execute_delete()
|-if DROP   -> _execute_drop_table()
|_ if ALTER  -> _execute_alter_table()
```

4.2 CREATE TABLE

```
_execute_create_table(stmt)
|-convert AST columns to Column objects
|-create Table object
|-call storage.create_table(table)
|_ write schema.json
```

4.3 INSERT INTO

```
_execute_insert(stmt)
|-rows = [{BookID: 1, Title: 'Data Mining', Price: 700}]
|-validate foreign keys via _validate_foreign_keys()
|-storage.insert_data(table, rows)
|-rebuild PK index
|_ return QueryResult(affected=1)
```

4.4 SELECT (Simple)

```
_execute_select(stmt)
|-load base table
|-apply WHERE via _apply_where_clause()
|-apply ORDER / DISTINCT if needed
|-_select_columns()
|_ return QueryResult(rows)
```

4.5 SELECT (JOIN)

```
_execute_join(stmt)
|-base_data = storage.get_table(MEMBERS)
|-join ISSUE_DETAILS I via _inner_join()
|-join BOOKS B via _inner_join()
|-apply WHERE and ORDER
|-select projected columns
|_ return QueryResult(data)
```

4.6 UPDATE

```
_execute_update(stmt)
|-table = storage.get_table(stmt.table)
|-for row in table.data:
|     if _evaluate_where_clause(row):
|         apply set_clause
|-storage._save_table_data()
|_ return QueryResult(affected_rows)
```

4.7 DELETE

```
_execute_delete(stmt)
|-table = storage.get_table(stmt.table)
|-remove rows satisfying where_clause
|-save table data
|_ return QueryResult(affected_rows)
```

4.8 DROP TABLE

```
_execute_drop_table(stmt)
|-remove from schema
|-delete JSON file
|-remove index directory
|_ confirm deletion
```

5 Storage Engine

The StorageManager handles reading/writing table data and schema management in JSON.

```
create_table()
|-schema.add_table()
|-save schema.json
|_ create empty table.json
```

```
insert_data()
|-load table.json
|-validate row & foreign keys
|-append to table.data
|-save updated JSON
|_ rebuild index
```

6 Indexed Storage Manager

The IndexedStorageManager enhances performance using on-disk indexes.

Index Workflow

```
create_table()
|_ _ensure_pk_index()
insert_data()
|_ _rebuild_index()
update_data(), delete_data()
|_ _rebuild_index()
```

Indexed SELECT

```
select_data()
|-detect WHERE column = value
|-load index JSON
|-use PK set to filter rows
|_ return optimized result
```

7 Error Handling System

Errors inherit from CoreDBError.

```
CoreDBError
|-SQLSyntaxError
|-StorageError
|-TableNotFoundError
|_ ColumnNotFoundError
```

They propagate through QueryExecutor.execute() and are wrapped in QueryResult(success=False, error=...).

8 Foreign Key Validation

Before insertions, CoreDB enforces referential integrity.

```
_validate_foreign_keys(table, row)
|-for fk_col in table.columns:
|     -> load referenced_table
|     -> ensure referenced_column exists
|     -> check fk_value presence
|_ raise StorageError on violation
```

9 End-to-End Execution Examples

9.1 Example 1: INSERT and SELECT

```
CREATE TABLE BOOKS(BookID INT PRIMARY KEY, Title TEXT, Price FLOAT);
INSERT INTO BOOKS VALUES (1, 'Data Mining', 700);
SELECT * FROM BOOKS;
```

Execution path:

```
Lexer -> Parser -> Executor -> Storage -> JSON -> Result
```

Result:

```
[{'BookID': 1, 'Title': 'Data Mining', 'Price': 700}]
```

9.2 Example 2: UPDATE and DELETE

```
UPDATE BOOKS SET Price = 800 WHERE BookID = 1;
DELETE FROM BOOKS WHERE Price > 700;
```

Executor functions invoked:

```
_execute_update() -> modifies row -> save JSON
_execute_delete() -> filters JSON -> save JSON
```

9.3 Example 3: DROP TABLE

```
DROP TABLE BOOKS;
```

```
_execute_drop_table()
|-schema.drop_table('BOOKS')
|-delete BOOKS.json
|_ confirm removal
```