UNINORTE

Pró-Reitoria Acadêmica
Escola de Ciências Exatas e Tecnologia
Curso de Ciência da Computação

Natanael Santos da Silva – 03323776

Relatório: Projeto de Banco de Dados

Manaus
2025

Contents

1 Introdução

Os bancos de dados relacionais são pilares fundamentais da computação moderna, permitindo o armazenamento, gerenciamento e recuperação eficiente de informações em sistemas que variam de aplicações simples a plataformas complexas de inteligência artificial. Nesse contexto, o SQLite, um banco de dados leve e embarcado, destaca-se por sua simplicidade e robustez, sendo amplamente utilizado em projetos acadêmicos e profissionais.

Este relatório documenta o desenvolvimento de um sistema de gerenciamento de pedidos implementado com SQLite, como parte de um trabalho acadêmico da disciplina de Banco de Dados. O projeto envolveu a criação de tabelas, triggers, views, índices e chaves estrangeiras, além da implementação de testes automatizados com pytest para validar a funcionalidade do sistema. O objetivo foi explorar conceitos fundamentais de bancos de dados relacionais, avaliar a eficácia de cada componente e propor melhorias para aplicações futuras.

A metodologia incluiu o uso de Python 3.8, SQLite para o gerenciamento do banco, pytest para testes automatizados e LaTeX para a documentação do relatório. O projeto foi versionado no GitHub, garantindo organização e rastreabilidade das alterações.

2 Métodos

Descrevo aqui os métodos utilizados para desenvolver e testar o sistema de banco de dados, organizados por componente.

2.1 Criação de Tabelas

Três tabelas foram criadas: `users`, `orders` e `logs`. A tabela `users` armazena informações de usuários, `orders` registra pedidos associados a usuários, e `logs` mantém um histórico de inserções. O script `create_tables.sql` foi usado para definir a estrutura, com chaves primárias e estrangeiras para garantir integridade.

2.2 Trigger

Uma trigger chamada `order_insert` foi implementada para registrar automaticamente em `logs` cada inserção na tabela `orders`. O script `create_trigger.sql` define a trigger, que captura o `user_id` do pedido e o timestamp da operação.

2.3 View

A view `user_orders` foi criada para combinar dados de `users` e `orders` usando um INNER JOIN. O script `create_view.sql` define a view, facilitando consultas que exibem informações de usuários e seus pedidos.

2.4 Índice

Um índice chamado `idx_name` foi criado no campo `name` da tabela `users` para otimizar consultas por nome. O script `create_index.sql` define o índice, e sua eficácia foi testada em cenários de busca.

2.5 Chave Estrangeira

A tabela `orders` inclui uma chave estrangeira (`user_id`) que referencia a chave primária (`id`) de `users`. A integridade referencial foi configurada com `FOREIGN KEY` e testada para evitar registros órfãos.

2.6 Testes Automatizados

Os testes foram implementados com pytest na pasta `tests/`, usando fixtures definidas em `conftest.py` para gerenciar conexões SQLite em memória. Seis arquivos de teste foram criados:

- `test_update.py`: Testa a atualização de nomes em `users`.
- `test_view.py`: Valida a view `user_orders`.
- `test_trigger.py`: Verifica o funcionamento da trigger `order_insert`.
- `test_foreign_key.py`: Testa a integridade referencial.
- `test_index.py`: Avalia o uso do índice `idx_name`.
- `test_additional.py`: Testa inserção, deleção, joins, transações e concorrência.

3 Resultados

Os testes foram executados com sucesso, validando a funcionalidade de cada componente. Abaixo estão os principais resultados:

3.1 Testes de Atualização

O teste `test_update_user` confirmou que o nome de um usuário pode ser atualizado corretamente:

- Entrada: Inserir usuário com nome “Alice”, atualizar para “Alicia”.
- Saída: A consulta retornou “Alicia”.

3.2 Testes de View

O teste `test_view` verificou que a view `user_orders` retorna dados corretos:

- Entrada: Inserir usuário “Bob” e pedido associado.
- Saída: A view retornou o nome “Bob” e o pedido correspondente.

3.3 Testes de Trigger

O teste `test_trigger` confirmou que a trigger `order_insert` registra logs:

- Entrada: Inserir um pedido para `user_id=1`.
- Saída: A tabela `logs` continha um registro com `user_id=1` e timestamp.

3.4 Testes de Chave Estrangeira

O teste `test_foreign_key` validou a integridade referencial:

- Entrada: Tentar inserir um pedido com `user_id` inexistente.
- Saída: Erro de violação de chave estrangeira, como esperado.

3.5 Testes de Índice

O teste `test_index` confirmou que o índice `idx_name` é usado:

- Entrada: Consultar `users` por `name="Alice"`.
- Saída: O plano de execução indicou o uso do índice.

3.6 Testes Adicionais

O teste `test_additional` validou inserção, deleção, joins, transações e concorrência, com todas as operações funcionando conforme esperado.

4 Análise

Os resultados demonstram que o sistema é funcional para cenários simples, mas algumas limitações foram observadas:

- **Tabelas e Chaves Estrangeiras**: A integridade referencial foi eficaz, mas o sistema não suporta relacionamentos complexos, como múltiplas chaves estrangeiras.
- **Trigger**: A trigger `order_insert` é limitada a inserções, sem suporte para atualizações ou deleções.
- **View**: A view `user_orders` é eficiente, mas não inclui filtros adicionais para consultas específicas.
- **Índice**: O índice `idx_name` melhora a performance, mas sua eficácia depende do volume de dados.
- **Testes**: Os testes cobrem casos básicos, mas não avaliam cenários de alta concorrência ou grandes volumes de dados.

5 Recomendações

Com base nos resultados e análises, sugerimos as seguintes melhorias:

- **Tabelas**: Suportar relacionamentos mais complexos, como tabelas intermediárias para relações muitos-para-muitos.
- **Trigger**: Implementar triggers para atualizações e deleções, registrando todas as alterações em `logs`.

- **View**: Adicionar parâmetros à view para filtrar resultados dinamicamente.
- **Índice**: Testar índices compostos para otimizar consultas com múltiplos critérios.
- **Testes**: Incluir testes de estresse com grandes volumes de dados e concorrência intensa.

6 Conclusão

Este projeto proporcionou uma compreensão prática dos conceitos de bancos de dados relacionais, abrangendo desde a criação de estruturas de dados até a validação automatizada de funcionalidades. A implementação do sistema de gerenciamento de pedidos com SQLite demonstrou a importância de componentes como chaves estrangeiras para garantir integridade, triggers para automação, views para consultas simplificadas e índices para desempenho. Os testes com pytest validaram a funcionalidade de cada componente, destacando a robustez do sistema para cenários simples.

As principais contribuições incluem a aplicação prática de conceitos teóricos, a documentação detalhada do processo e a identificação de limitações que podem orientar melhorias futuras. Apesar de eficaz, o sistema é limitado a casos simples, indicando a necessidade de expansão para suportar cenários mais complexos. Para trabalhos futuros, recomendo a implementação de triggers adicionais, views parametrizadas, índices compostos e testes de estresse, além de explorar outros bancos de dados, como PostgreSQL, para comparar desempenho.

Em resumo, o projeto consolidou o aprendizado sobre bancos de dados relacionais, fornecendo uma base sólida para o desenvolvimento de sistemas mais avançados no futuro.

7 Anexos

7.1 Criação de Tabelas

O script `create_tables.sql` define as tabelas do sistema:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE orders (
    id INTEGER PRIMARY KEY,
    user_id INTEGER,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE logs (
    id INTEGER PRIMARY KEY,
    user_id INTEGER,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

7.2 Teste de Atualização

Trecho do `test_update.py`:

```
def test_update_user(db_connection, setup_database):
    cursor = db_connection.cursor()
    cursor.execute("INSERT INTO users (name) VALUES ('Alice')")
    db_connection.commit()
    cursor.execute("UPDATE users SET name = 'Alicia' WHERE name = 'Alice'")
    db_connection.commit()
    cursor.execute("SELECT name FROM users WHERE name = 'Alicia'")
    result = cursor.fetchone()
    cursor.close()
    assert result is not None, "O nome atualizado deveria existir."
    assert result[0] == 'Alicia', "O nome deveria ser 'Alicia'."
```

7.3 Saída de Testes

Execução dos testes com pytest:

```
$ pytest tests/
========================= test session starts ==========================
platform win32 -- Python 3.8.10, pytest-7.4.4
collected 6 items

tests/test_update.py .                                            [ 16%]
tests/test_view.py .                                              [ 33%]
tests/test_trigger.py .                                           [ 50%]
tests/test_foreign_key.py .                                       [ 66%]
tests/test_index.py .                                             [ 83%]
tests/test_additional.py .                                        [100%]

========================== 6 passed in 0.12s ===========================
```