CS315 Project

April 24, 2025

Departmental Store Database Application

Sajal Jain	210903	`jsajal21@iitk.ac.in`
Siddhant Singhai	211028	`ssinghai21@iitk.ac.in`
Akshat Mehta	210092	`akshatm21@iitk.ac.in`
Abhishek Choudhary	210037	`cabhishek21@iitk.ac.in`
Aryan Thada	210205	`aryant21@iitk.ac.in`

Link to code:- `https://github.com/JainSajal23/CS315/tree/main`

Abstract

We present a high-performance, multi-threaded departmental store database application developed using Python and Oracle Database. Leveraging the `oracledb` module and a configurable connection pool, the system simulates realistic workloads, including customer creation, order placement with randomized item generation, updates, deletions, and summary queries, while ensuring ACID compliance. Detailed logging and tracing capture operational metrics (e.g., transaction counts, pass/fail rates) and fine-grained SQL execution context, enabling deadlock detection, session monitoring, and performance profiling. Results demonstrate robust handling of concurrent transactions over configurable durations or iteration counts, with complete transactional integrity and real-time visibility into pool utilization, session activity, and lock contention.

1 Motivation and Problem Statement

Retail enterprises face the challenge of processing high volumes of concurrent transactions—such as order entry and inventory updates—while ensuring accuracy, consistency, and responsiveness. Key challenges include:

- **Order Processing:** Support atomic, multi-step order workflows (e.g., item insertions, order header creation) with rollback on failure.
- **Customer Data Integrity:** Safeguard personal and financial details using parameterized SQL and transaction isolation.
- **High Concurrency:** Scale to 50–100 pooled connections and multiple worker threads to emulate real-world loads without performance degradation.
- **Operational Observability:** Provide dual logging (summary and trace) and real-time monitoring to quickly detect issues like deadlocks or unexpected errors.

2 Methodology

We adopted a layered, configurable architecture to ensure maintainability, scalability, and comprehensive observability.

2.1 Layered Architecture

- **Database Layer (`db.py`):** Manages Oracle connectivity—standalone or via a 50–100 connection pool.
- **Business Logic Layer (`action.py`):** Implements 29 granular functions for CRUD operations on `CUSTOMERS`, `ORDERS`, and `ITEMS`, including helper routines for randomized data generation and robust error handling (e.g., deadlock retries).
- **Logging and Tracing Layer (`logger_tracer.py`):** Configures two independent loggers: a *monitor logger* (summary info/warning/error) and a *trace logger* (detailed SQL context, debug/error traces), with all parameters driven by `db.config`.
- **Workload Management Layer (`main.py`):** Reads test-case weights and concurrency settings from `db.config`, dispatches weighted transactions via multiple threads, and aggregates success/failure metrics with periodic snapshots every 5 seconds.

2.2 Thread-Based Concurrency Model

- A configurable number of worker threads (`num_threads`) executes a shuffled list of operations based on weighted counts.
- A dedicated logging thread ensures non-blocking persistence of aggregated transaction counts and queued errors every 5 seconds.

2.3 Robust Error Handling

- Each operation uses `try/except` blocks with specific handling for `oracledb.DatabaseError` to classify errors (e.g., ORA-00001 duplicates, ORA-00060 deadlocks).
- Transactional operations explicitly call `connection.begin()`, `connection.commit()`, and `connection.rollback()` to maintain atomicity.

3 Implementation and Results

The implementation comprises four Python files and one configuration file:

- **`db.py`:** Manages database connections, pooling, and monitoring (e.g., session monitoring, lock detection).
- **`main.py`:** Orchestrates workload, schedules tasks, and manages threading and logging.
- **`action.py`:** Contains business logic for operations like adding customers, placing orders, and updating records.
- **`logger_tracer.py`:** Tracks selected test cases and logs results in a structured table format.
- **`db.config`:** Central to the application, classifying settings into four types:
    a. **Database Connection Pool Settings** ([pool]): Configures Oracle connectivity (e.g., `Pool`, `user`, `password`, `dsn`) and pool parameters (e.g., `min=50`, `max=100`, `increment=10`, `timeout=15`).

b. **Transaction Execution Settings** ([transactions]): Defines workload parameters (e.g., `doThreading=True`, `num_threads=10`, `iterations=100`, `duration_mins=0`).

c. **Test Case Weight Settings** ([test_cases]): Assigns weights to prioritize test cases (e.g., `orders_ofcust_custo` `summarise_orders_forcust=1`, `summarise_orders_forcust_customdate=1`, others=0).

d. **Logging and Tracing Settings** ([address], [log_file_handle], [trace_file_handle]): Specifies log/-trace file directories (e.g., `log_file_direc`) and logging parameters (e.g., `log_level=10`, `trace_format`).

3.1 Database Schema and Operations

The system uses six key tables: `CUSTOMERS`, `ORDERS`, `ITEMS`, `PRODUCTS`, `WAREHOUSES`. It implements CRUD operations for all entities, with transaction control for multi-step operations.

3.2 Key Transactions

Add New Customer:

1. `gen_new_customer_id()`: Fetches `CUSTOMERS_SEQ.NEXTVAL` with 10 retries.
2. `chckIf_cId_present()`: Ensures uniqueness.
3. `insert_customer()`:

```
INSERT INTO CUSTOMERS VALUES (:CUSTID, :NAME, ..., :CREDITLIM)
```

Commits or rolls back with `traceInfo`/`traceError`.

Place Order:

1. New order ID via `ORDERS_SEQ`.
2. `insert_items()` loops:
   - Retrieves warehouse (`SELECT WHID`).
   - Retrieves price (`SELECT PRICE FROM PRODUCTS`).
   - Inserts into `ITEMS`.
3. `insert_order()`:

```
INSERT INTO ORDERS VALUES (:ORDID, :CUSTID, ...)
```

Full commit/rollback block.

Delete Customer & Cascade: • Deletes `ITEMS` → `ORDERS` → `CUSTOMERS`.

Order Summary: • Aggregates bill with subquery:

```
SELECT SUM(PRICE*QTY) FROM ITEMS WHERE ITEMS.ORDID = ORDERS.ORDID
```

Formatted with `tabulate` into trace logs.

3.3 Error Handling and Transaction Integrity

Robust error handling and transaction integrity are ensured via commit/rollback. The system includes trace calls and log calls to monitor behavior.

Results demonstrate successful workload execution for a 1-minute duration (as configured). The system handled concurrent transactions without significant failures, with logs capturing successes and errors. Monitoring functions provided clear visibility into session activity and lock conditions, with sample output including tabulated session data and lock reports, confirming the system's ability to manage concurrent workloads.

4 Discussion and Limitations

The application effectively simulates a departmental store's query-intensive workload, with `db.config` enabling flexible configuration. Connection pooling and multi-threading ensure scalability, while monitoring aids performance analysis. A significant challenge was overcome deadlock issue in this multi-threaded simulation, which was addressed through query optimization.

4.1 Deadlock Analysis and Resolution

Consider this example.
The original SQL query used in the `deleting_cust` function was:

```
delete_item = """ DELETE FROM ITEMS WHERE ORDID IN (
                    SELECT ORDID FROM ORDERS WHERE CUSTID=:ID
                ) """
delete_order = """ DELETE FROM ORDERS WHERE CUSTID = :ID """
delete_cust = """ DELETE FROM CUSTOMERS WHERE CUSTID = :ID """
```

This query led to a deadlock scenario, where two threads (T1 and T2) executed concurrently. The process unfolded as follows:

- **Step 1:** T1 acquired a shared lock on `ORDERS[101]` to read, while T2 acquired a shared lock on `ORDERS[102]`.
- **Step 2:** T1 deleted from `ITEMS` (locking `ITEMS[ORDID from 101]`), and T2 attempted the same for `ITEMS[ORDID from 102]` but have to wait.
- **Step 3:** T1 attempted an exclusive lock on `ORDERS[101]` to delete but was blocked by T2's shared lock (held due to its subquery).
  Thus, T1 have to wait for T2 to release lock from `ORDERS` and T2 have to wait for T1 to release lock from `ITEMS`, leading to deadlock.

To mitigate this, the query was revised to use the `EXISTS` clause:

```
del_sql = """
    DELETE FROM ITEMS I
    WHERE EXISTS (
        SELECT 1 FROM ORDERS O
        WHERE I.ORDID = O.ORDID AND O.CUSTID = :ID
    )
    """
```

This new approach leverages Oracle's row-by-row evaluation of `EXISTS`, offering the following benefits:

- **Reduced Lock Duration:** `EXISTS` short-circuits early, minimizing the time shared locks are held.
- **Smaller Lock Scope:** Locks are applied only to matching rows in `ORDERS`, avoiding the entire table.
- **Lower Blocking Risk:** Short-lived shared locks prevent contention with subsequent exclusive locks.
- **Avoids Intermediate Dependencies:** Deletes occur as checks are made, eliminating materialization of `ORDID` sets.

The locking schedule for a transaction (e.g., T1 with `CUSTID = 101`) involves acquiring shared locks (`S-lock`) on `ORDERS` rows only during the `EXISTS` check, releasing them quickly, followed by exclusive locks (`X-lock`) for deletions, significantly reducing deadlock risk.

4.2 Limitations

Despite this improvement, the following limitations persist:

- **Limited Test Case Scope:** Zero weights for write operations (e.g., `addnewcustomer`, `place_order`) restrict testing of data modification tasks.
- **Scalability Testing:** Testing with 100 iterations and 10 threads may not reflect peak loads.
- **Static Configuration:** `db.config` lacks runtime updates.

Future work could expand test case coverage, enhance scalability testing, and enable dynamic configuration.

5 ER Diagram

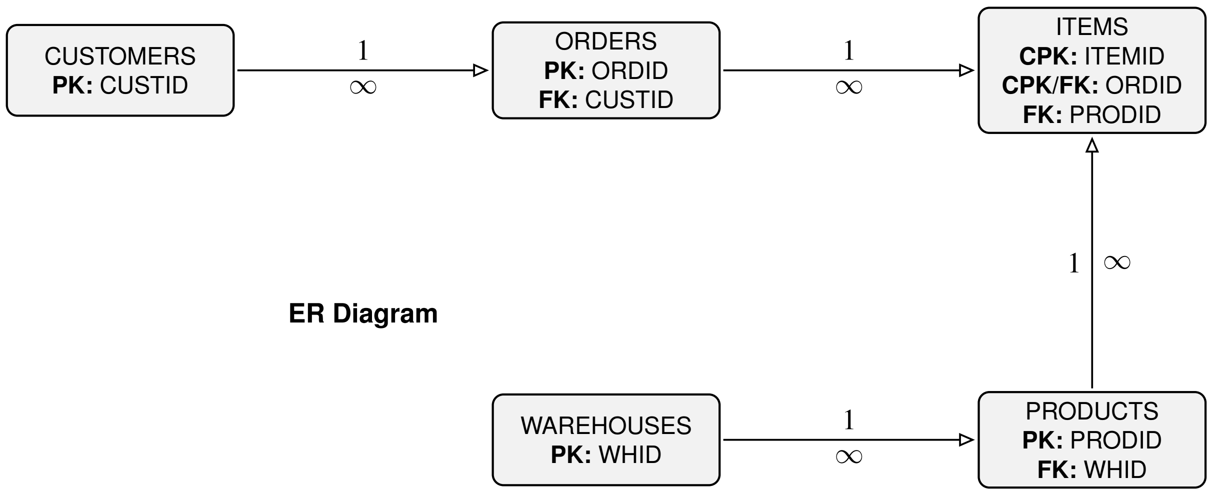

Figure 1: Entity-Relationship Diagram of the Departmental Store Database

6 Contributions

- **Sajal Jain:** Designed the overall architecture, implemented `db.py` for database connectivity, worked upon `action.py` focusing on query optimization.
- **Siddhant Singhai:** Developed `main.py`, including task scheduling, threading, and workload orchestration.
- **Akshat Mehta:** Implemented `action.py`, focusing on business logic for customer and order operations.
- **Abhishek Choudhary:** Implemented logging and tracing mechanisms in `logger_tracer.py` and worked upon `SQL` queries.
- **Aryan Thada:** Tested the application, validated monitoring outputs, and documented results.