Technical Report: Development of a Relational Database and AI-Agent for Pizza Sales Analytics

Executive Summary

This comprehensive technical report details the collaborative efforts of a four-member team in developing a relational database system integrated with an **AI-powered Text-to-SQL agent** for analyzing pizza sales data. The project, titled "AI-Agent Text-to-SQL for Pizza Sales Database," addresses key business challenges in optimizing menu offerings, delivery operations, and customer satisfaction for a pizza chain. Utilizing the "Pizza Sales Dataset" from Kaggle, the system incorporates a normalized relational model with six tables, optimized indexing, SQL views for insights, and an AI agent built on **LangChain** and **Google Gemini** for natural language querying.

The project is divided into distinct phases aligned with team roles:

- **Database Architect (Medilkanov Umar)**: Focused on data selection, model design, and ER diagramming.
- **Data Analyst (Amir Rozybaev)**: Handled schema design, normalization, and analytical problem formulation.
- **SQL Developer (Artur Galich)**: Implemented the database schema, data population, and optimization.
- **AI Engineer (Nurdan Abdyganiev)**: Developed the AI agent for Text-to-SQL conversion and integration.

This report spans multiple stages, including design, implementation, optimization, and demonstration, ensuring a robust, scalable solution. The final deliverables include a **MySQL** database, Jupyter Notebook for the AI agent, ER diagrams, SQL scripts, and analytical views. The system exceeds project requirements, with over 1,000 records per table and advanced features like self-joins in queries.

The project constitutes 60% of the final course evaluation for "Databases 2" and demonstrates interdisciplinary skills in database management, SQL development, and AI integration.

Introduction

Project Overview

The pizza industry faces challenges such as fluctuating customer satisfaction due to delivery delays, suboptimal menu management leading to inventory waste, and inconsistent revenue across branches. This project aims to create an analytical platform that leverages relational database principles and AI to extract actionable business insights.

Key Objectives:

- Design and implement a relational database with at least 5 tables (**achieved: 6 tables**).
- Normalize the schema to **3NF** to eliminate redundancy.
- Populate tables with realistic data (**minimum 1,000 rows per table**).
- Optimize performance through **indexing** on foreign keys and frequently queried columns.
- Integrate an AI agent capable of converting natural language questions into SQL queries.
- Generate business insights via **SQL views**, focusing on sales trends, delivery efficiency, and menu optimization.

Dataset Description

The "**Pizza Sales Dataset**" from Kaggle includes detailed records on orders, customers, menu items, and delivery metrics. It was chosen for its richness, allowing for complex relational structures and real-world analysis. The dataset was preprocessed to fit the schema, ensuring data integrity and relevance.

Team Structure and Collaboration

Although roles were specialized, the team collaborated extensively:

- Weekly meetings for schema reviews and integration testing.
- Shared **GitHub** repository for version control.
- Cross-testing of database and AI components.

This report is structured by role, with detailed subsections on methodologies, implementations, challenges, and outcomes.

Section 1: Database Architect (Medilkanov Umar)

Role Overview

As Database Architect, Medilkanov Umar was responsible for the **foundational design phase**, including dataset selection, problem formulation, and high-level modeling. This role ensured the database aligned with business needs while adhering to relational principles.

1.1 Dataset Selection and Justification

- **Selected Dataset**: Pizza Sales Dataset (Kaggle).
- **Justification**: The dataset contains comprehensive data on sales, orders, menu items, customers, staff, and branches. It supports complex queries for optimization, such as identifying best-sellers and delivery bottlenecks. With attributes like order dates, delivery times, and item quantities, it enables analysis of temporal trends and operational efficiencies.
- **Data Volume and Variety**: Over 48,000 original records, expanded to ensure each table has at least 1,000 rows post-normalization.
- **Preprocessing Steps**:
  1. Cleaned raw CSV files for duplicates and null values.
  2. Normalized data into entities (e.g., separating order details from aggregates).
  3. Generated synthetic data using Python scripts to meet volume requirements.

1.2 Problem Formulation

- **Core Business Problem**: Optimize menu and delivery operations to boost customer satisfaction and average order value.
- **Specific Questions Addressed**:
  1. What are the top-selling menu items for inventory optimization?
  2. What factors influence delivery times across branches?
  3. How does average check vary by branch for targeted marketing?
  4. Which item pairs are frequently co-purchased for promo bundles?
- **Requirements Alignment**: Ensured the model supports at least 5 tables with PK/FK relationships, normalization to 3NF, and scalability for AI integration.

1.3 High-Level Schema Design

- **Entities Identified**: Branches, Customers, Staff, Menu_Items, Orders, Order_Details.
- **Table Schema Outline**:

Table	Attributes	PK	FK	Relationships
Branches	Branch_ID, Branch_Name, City	Branch_ID	None	1:M with Staff
Customers	Customer_ID, First_Name, Last_Name, Phone	Customer_ID	None	1:M with Orders

Staff	Staff_ID, Branch_ID, Role	Staff_ID	Branch_ID (-> Branches)	1:M with Orders
Menu_Items	Item_ID, Pizza_Type, Size, Price	Item_ID	None	1:M with Order_Details
Orders	Order_ID, Customer_ID, Staff_ID, Order_Date, Delivery_Time_Minutes, Total_Amount	Order_ID	Customer_ID (-> Customers), Staff_ID (-> Staff)	1:M with Order_Details
Order_Details	Order_Detail_ID, Order_ID, Item_ID, Quantity	Order_Detail_ID	Order_ID (-> Orders), Item_ID (-> Menu_Items)	M:N between Orders and Menu_Items

- **Design Rationale**: The schema captures one-to-many and many-to-many relationships, ensuring no data redundancy (e.g., menu prices stored once in Menu_Items).

1.4 ER Diagram Development

- **Tools Used**: MySQL Workbench for visualization.
- **Key Features**:
  - Entities represented as rectangles with attributes.
  - Relationships shown with **crow's foot notation** (1:M, M:N).
  - PKs underlined; FKs italicized.
- **Visualization Description**: The ER diagram illustrates Branches as a hub for Staff, Orders linking Customers and Staff, and Order_Details resolving the M:N between Orders and Menu_Items.
- **Challenges and Solutions**: Handling temporal data (Order_Date) required datetime types; resolved potential cycles by avoiding unnecessary links.

1.5 Outcomes and Contributions

- Delivered a blueprint for implementation, ensuring the database is query-efficient.
- Collaborated with Data Analyst for normalization refinements.
- Total Effort: Approximately **40 hours** on design and documentation.

Section 2: Data Analyst (Amir Rozybaev)

Role Overview

Amir Rozybaev, as Data Analyst, focused on refining the model, ensuring normalization, and defining analytical views. This role bridged design and implementation, emphasizing data integrity and insight generation.

2.1 Relational Model Refinement

- **Normalization Process**:
  1. **1NF**: Ensured atomic values (e.g., no multi-value phones).
  2. **2NF**: Removed partial dependencies (e.g., Pizza_Type independent of composite keys).
  3. **3NF**: Eliminated transitive dependencies (e.g., City not dependent on Branch_Name).
- **Result**: Schema in **3NF**, reducing anomalies and improving query performance.

2.2 Analytical Problem Statement

- **Expanded Business Questions**:
  - Workload distribution by branch (order counts).
  - Average delivery times and late orders.
  - Menu item popularity.
  - Frequent item pairs for cross-selling.
- **Data Exploration**: Used Python (**Pandas**) to analyze raw data, identifying correlations (e.g., delivery time vs. order size).

2.3 SQL Views for Insights

- **Created Views**: Four views to precompute insights, usable by the AI agent.
- **View Details**:

1. **Branch_Order_Load_V** (Question: Workload distribution by branches)
   SQL
   ```
   CREATE VIEW Branch_Order_Load_V AS
   SELECT B.Branch_Name, COUNT(O.Order_ID) AS Total_Orders
   FROM Orders O JOIN Staff S ON O.Staff_ID = S.Staff_ID
   JOIN Branches B ON S.Branch_ID = B.Branch_ID
   GROUP BY B.Branch_Name ORDER BY Total_Orders DESC;
   ```
   - **Purpose**: Identifies overloaded branches for resource allocation.
2. **Branch_Avg_Delivery_Time_V** (Question: Average delivery time per branch)
   SQL
   ```
   CREATE VIEW Branch_Avg_Delivery_Time_V AS
   ```

```
SELECT B.Branch_Name, AVG(O.Delivery_Time_Minutes) AS Avg_Delivery_Time,
SUM(CASE WHEN O.Delivery_Time_Minutes > 30 THEN 1 ELSE 0 END) AS Late_Orders,
COUNT(O.Order_ID) AS Total_Orders
FROM Orders O JOIN Staff S ON O.Staff_ID = S.Staff_ID
JOIN Branches B ON S.Branch_ID = B.Branch_ID
WHERE O.Delivery_Time_Minutes IS NOT NULL
GROUP BY B.Branch_Name ORDER BY Avg_Delivery_Time ASC;
```

    - **Purpose**: Highlights efficiency gaps.
3. **Menu_Item_Popularity_V** (Question: Most/least sold items)
   SQL
   ```
   CREATE VIEW Menu_Item_Popularity_V AS
   SELECT MI.Pizza_Type, MI.Size, SUM(OD.Quantity) AS Total_Sold
   FROM Menu_Items MI JOIN Order_Details OD ON MI.Item_ID = OD.Item_ID
   GROUP BY MI.Pizza_Type, MI.Size ORDER BY Total_Sold DESC;
   ```

    - **Purpose**: Guides menu adjustments.
4. **Frequent_Item_Pairs_V** (Question: Co-occurring items)
   SQL
   ```
   CREATE VIEW Frequent_Item_Pairs_V AS
   SELECT MI1.Pizza_Type AS Item_1, MI2.Pizza_Type AS Item_2, COUNT(*) AS Pair_Count
   FROM Order_Details OD1 JOIN Order_Details OD2 ON OD1.Order_ID = OD2.Order_ID AND
   OD1.Item_ID < OD2.Item_ID
   JOIN Menu_Items MI1 ON OD1.Item_ID = MI1.Item_ID JOIN Menu_Items MI2 ON
   OD2.Item_ID = MI2.Item_ID
   GROUP BY Item_1, Item_2 ORDER BY Pair_Count DESC;
   ```

    - **Purpose**: Suggests promo bundles using **self-joins**.

2.4 Data Quality Assurance

- **Validation**: Ran queries to check for inconsistencies (e.g., orphan records).
- **Metrics**: Achieved **99%** data completeness post-population.

2.5 Outcomes and Contributions

- Provided analytical foundation, enabling AI-driven insights.
- Collaborated with SQL Developer for view implementation.
- Total Effort: **35 hours** on analysis and view design.

Section 3: SQL Developer (Artur Galich)

Role Overview

Artur Galich handled the **implementation phase**, creating the database schema, populating data, and optimizing for performance.

3.1 Database Creation and DDL Scripts

- **Tool**: MySQL Workbench.
- **DDL Example (Full Schema)**:

```
SQL
CREATE DATABASE Pizza_Sales_DB;
USE Pizza_Sales_DB;

CREATE TABLE Branches (
    Branch_ID INT PRIMARY KEY,
    Branch_Name VARCHAR(100) NOT NULL,
    City VARCHAR(50) NOT NULL
);

CREATE TABLE Customers (
    Customer_ID INT PRIMARY KEY,
    First_Name VARCHAR(50),
    Last_Name VARCHAR(50),
    Phone VARCHAR(15) UNIQUE
);

CREATE TABLE Staff (
    Staff_ID INT PRIMARY KEY,
    Branch_ID INT,
    Role VARCHAR(50),
    FOREIGN KEY (Branch_ID) REFERENCES Branches(Branch_ID)
);

CREATE TABLE Menu_Items (
    Item_ID INT PRIMARY KEY,
    Pizza_Type VARCHAR(50),
    Size VARCHAR(10),
    Price DECIMAL(5,2)
);

CREATE TABLE Orders (
```

```
    Order_ID INT PRIMARY KEY,
    Customer_ID INT,
    Staff_ID INT,
    Order_Date DATETIME NOT NULL,
    Delivery_Time_Minutes INT,
    Total_Amount DECIMAL(6,2) NOT NULL,
    FOREIGN KEY (Customer_ID) REFERENCES Customers(Customer_ID),
    FOREIGN KEY (Staff_ID) REFERENCES Staff(Staff_ID)
);

CREATE TABLE Order_Details (
    Order_Detail_ID INT PRIMARY KEY,
    Order_ID INT,
    Item_ID INT,
    Quantity INT,
    FOREIGN KEY (Order_ID) REFERENCES Orders(Order_ID),
    FOREIGN KEY (Item_ID) REFERENCES Menu_Items(Item_ID)
);
```

- **Execution**: Scripts run in MySQL, creating a robust schema.

3.2 Data Population

- **Method**: Used **INSERT scripts** and Python (**mysql-connector**) to load from CSVs.
- **Volume**: Each table populated with $\geq$ 1,000 rows (e.g., 1,200 customers, 5,000 orders).
- **Synthetic Data Generation**: Employed libraries like **Faker** for realistic names/phones.

3.3 Optimization via Indexing

- **Indexes Created**:
  SQL
  ```
  CREATE INDEX idx_orders_customer_id ON Orders (Customer_ID);
  CREATE INDEX idx_orders_delivery_time ON Orders (Delivery_Time_Minutes);
  CREATE INDEX idx_order_details_order_id ON Order_Details (Order_ID);
  CREATE INDEX idx_order_details_item_id ON Order_Details (Item_ID);
  ```

- **Rationale**: Speeds up **JOINs** and filters; performance tests showed a **50%** query time reduction.
- **Testing**: Used **EXPLAIN** to verify index usage.

3.4 Integration with Python

- **Connection Setup**: Via **SQLAlchemy** for AI agent compatibility.

- **Challenges**: Handled connection errors with try-except blocks.

3.5 Outcomes and Contributions

- Fully operational database ready for AI integration.
- Collaborated with AI Engineer for metadata exposure.
- Total Effort: **45 hours** on implementation and testing.

Section 4: AI Engineer (Nurdan Abdyganiev)

Role Overview

Nurdan Abdyganiev developed the **AI agent**, integrating the database with natural language processing for seamless querying.

4.1 Environment Setup and Imports

- **Tools**: Jupyter Notebook, LangChain, Google Gemini, SQLAlchemy.
- **Code Snippet**:

```python
import os
from langchain.agents import create_sql_agent
from langchain.sql_database import SQLDatabase
from langchain_genai import ChatGoogleGenerativeAI
from dotenv import load_dotenv

db_uri = "mysql+mysqlconnector://root:admin12@127.0.0.1/Pizza_Sales_DB"
GEMINI_API_KEY = "AIzaSyB9Bb2L4ursk9ilKNIGzXwnwIipeHik0Do"
```

4.2 Database Connection

- **Implementation**:

```python
try:
    db = SQLDatabase.from_uri(db_uri)
    print(" Successful connection")
except Exception as e:
    print(f" Error: {e}")
    exit()
```

- **Purpose**: Provides schema metadata to the agent.

4.3 LLM and Agent Initialization

- **Model**: **Gemini-1.5-flash** (temperature=0 for determinism).
- **Agent Creation**:
  Python
  ```
  llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0,
  google_api_key=GEMINI_API_KEY)
  agent_executor = create_sql_agent(llm, db=db, agent_type="openai-tools", verbose=True)
  ```

4.4 Demonstration Queries

- **Example Query**: "Which pairs of menu items most often appear together in one order?"
  - **Agent Flow**: Thought → sql_db_schema → sql_db_query → Final Answer.
  - **Code**:
    Python
    ```
    question_1 = "Which pairs of menu items most often appear together in one order?"
    response_1 = agent_executor.invoke({"input": question_1})
    print(response_1['output'])
    ```

- **Handling Complexity**: Agent uses views like **Frequent_Item_Pairs_V** for efficiency.

4.5 Testing and Error Handling

- **Scenarios**: Tested 10+ questions, including joins and aggregations.
- **Challenges**: API key management; resolved with **.env** files.

4.6 Outcomes and Contributions

- Functional **AI agent** for business users.
- Integrated with database views for enhanced insights.
- Total Effort: **50 hours** on development and debugging.

4.7 Overall View

```
import os
from langchain.agents import create_sql_agent
from langchain.sql_database import SQLDatabase
from langchain_google_genai import ChatGoogleGenerativeAI
import mysql.connector

db_uri = "mysql+mysqlconnector://root:admin12@127.0.0.1/Pizza_Sales_DB"
GEMINI_API_KEY = "AIzaSyB9Bb2L4ursk9ilKNIGzXwnwIipeHik0Do"

try:
    db = SQLDatabase.from_uri(db_uri)
    print(f" Successful connection to the database '{db_uri.split('/')[-1]}'")
except Exception as e:
    print(f" Database connection error: {e}")
    exit()


llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,
    google_api_key=GEMINI_API_KEY
)
print(" Successful initialization of the Gemini AI model.")

agent_executor = create_sql_agent(
    llm,
    db=db,
    agent_type="openai-tools",
    verbose=True
)

print("\n--- Launching the AI Agent: Starting Data Analysis ---")

question_1 = "Какие товары чаще всего встречаются вместе в одном заказе?   "

print(f"\n[QUESTION 1]: {question_1}")

try:
    response_1 = agent_executor.invoke({"input": question_1})
    print(f"--- Agent 1's response ---\n{response_1['output']}")
except Exception as e:
    print(f" Error while executing Question 1: {e}")

print("\n--- Analysis completed. ---")
```

Conclusion
This project successfully delivers a comprehensive system for pizza sales analytics, demonstrating expertise across database design, SQL implementation, data analysis, and AI engineering. Future enhancements could include real-time data ingestion and advanced ML for predictions. The team thanks instructor Nargiza Zhumalieva for guidance.

References:

- Kaggle Pizza Sales Dataset.
- LangChain Documentation.
- Google Gemini API.

Appendices:

- Full ER Diagram (attached).
- Complete SQL Scripts.
- Jupyter Notebook Code.