PostgreSQL Database Design & SQL Analysis Project

Portfolio case study from a graduate-level database/information management project for a Health Food Store & Café scenario.

Project Overview

This project designed, implemented, queried, optimized, and managed a PostgreSQL relational database for a combined retail health food store and café. The database supports products, café menu items, customers, employees, suppliers, orders, ingredients, inventory, and business reporting.

The summary below curates the strongest evidence from the original academic submission and presents it as a professional case study. The work is academic in context, but the database decisions are framed around practical information systems concerns: data integrity, maintainability, reporting, performance, and access control.

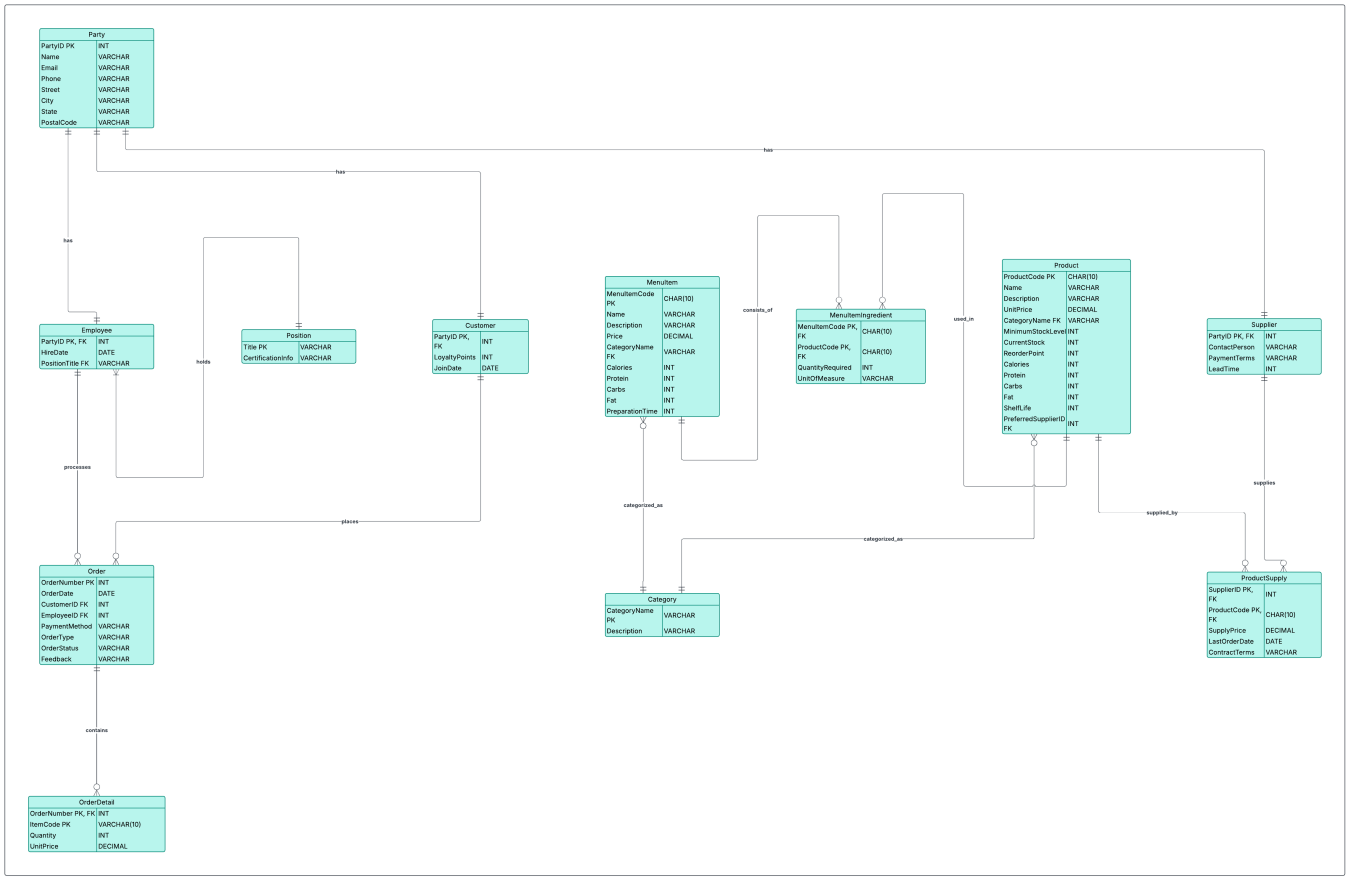
Database Design

Main entities include **Party, Customer, Employee, Supplier, Product, MenuItem, Order, OrderDetail, ProductSupply, MenuItemIngredient, Category,** and **Position**.

A key redesign centralized shared contact fields into a **Party** table. Customer, Employee, and Supplier records reference Party through `partyid`, reducing duplicated name, address, phone, and email fields and creating a cleaner source of truth.

The design also introduced a **Category** lookup table for Product and MenuItem classification. This avoids free-text category drift, supports consistent reporting, and leaves room for future category hierarchy without changing the core Product or MenuItem tables.

The redesign moved the schema toward normalization and 3NF by separating role-specific attributes, lookup values, and many-to-many relationships such as supplier-product supply and menu-item ingredients.



Original ERD from the project; zoom recommended for full schema details.

PostgreSQL Implementation

Selected implementation snippets show table design, primary keys, and referential integrity patterns used by the database.

```
CREATE TABLE party (
    partyid     INT          NOT NULL,
    name        VARCHAR(150) NOT NULL,
    email       VARCHAR(150) NOT NULL,
    phone       VARCHAR(50)  NOT NULL,
    street      VARCHAR(100) NOT NULL,
    city        VARCHAR(50)  NOT NULL,
    state       VARCHAR(50)  NOT NULL,
    postalcode  VARCHAR(20)  NOT NULL,
    CONSTRAINT party_pkey PRIMARY KEY (partyid)
);
```

```
CREATE TABLE product (
    productcode         CHAR(10)       NOT NULL,
    name                VARCHAR(100)   NOT NULL,
    unitprice           DECIMAL(10,2)  NOT NULL,
    categoryname        VARCHAR(50)    NOT NULL,
    currentstock        INT            NOT NULL,
    reorderpoint        INT            NOT NULL,
    preferredsupplierid INT,
    CONSTRAINT product_pkey PRIMARY KEY (productcode)
);
```

```
ALTER TABLE product
ADD CONSTRAINT fk_product_category
    FOREIGN KEY (categoryname)
    REFERENCES category (categoryname);

ALTER TABLE "order"
ADD CONSTRAINT fk_order_customer
    FOREIGN KEY (customerid)
    REFERENCES customer (partyid);
```

SQL Querying & Business Reporting

Product/category validation

Business need: Verify product category assignments and display category descriptions.

```
SELECT p.productcode, p.name AS product_name, p.unitprice,
       c.description AS category_description
FROM product p
INNER JOIN category c ON p.categoryname = c.categoryname
ORDER BY p.name ASC;
```

Demonstrates an INNER JOIN between product data and a lookup table.

Inventory forecasting

Business need: Identify total units sold per product for reorder planning.

```
SELECT od.itemcode AS product_code, pr.name AS product_name,
       SUM(od.quantity) AS total_units_sold
FROM orderdetail od
INNER JOIN product pr ON od.itemcode = pr.productcode
GROUP BY od.itemcode, pr.name
ORDER BY total_units_sold DESC;
```

Demonstrates aggregate reporting with GROUP BY and SUM.

SQL Querying & Business Reporting

Loyalty analytics

Business need: Understand customer order count and average order value.

```
SELECT c.partyid AS customer_id, p.name AS customer_name,
       COUNT(DISTINCT o.ordernumber) AS total_orders,
       ROUND(AVG(od.quantity * od.unitprice)::NUMERIC, 2) AS avg_order_value
FROM customer c
INNER JOIN "order" o ON c.partyid = o.customerid
INNER JOIN orderdetail od ON o.ordernumber = od.ordernumber
INNER JOIN party p ON c.partyid = p.partyid
GROUP BY c.partyid, p.name;
```

Demonstrates customer-level analytics using joins, COUNT, AVG, and calculated order values.

Sales line reporting

Business need: Provide finance with order date, customer, item, quantity, and line total.

```
SELECT o.ordernumber, o.orderdate, p.name AS customer_name,
       od.itemcode AS sold_item_code, pr.name AS sold_item_name,
       od.quantity, od.unitprice,
       (od.quantity * od.unitprice) AS line_total
FROM "order" o
INNER JOIN customer c ON o.customerid = c.partyid
INNER JOIN party p ON c.partyid = p.partyid
INNER JOIN orderdetail od ON o.ordernumber = od.ordernumber
LEFT JOIN product pr ON od.itemcode = pr.productcode
ORDER BY o.orderdate DESC, o.ordernumber;
```

Demonstrates a multi-table report across orders, customers, party/contact records, order details, and product details.

Category-based customer identification

Business need: Identify customers who purchased Supplements products.

```
SELECT DISTINCT p.name AS customer_name
FROM party p
WHERE p.partyid IN (
    SELECT o.customerid
    FROM "order" o
    WHERE o.ordernumber IN (
        SELECT od.ordernumber
        FROM orderdetail od
        WHERE od.itemcode IN (
            SELECT pr.productcode
            FROM product pr
            WHERE pr.categoryname = 'Supplements'
        )
    )
)
ORDER BY p.name;
```

Demonstrates nested subqueries for customer segmentation based on purchase behavior.

Optimization & Database Administration

The project added foreign key constraints to enforce referential integrity, then selected indexes based on join paths, filters, sorting, and likely reporting patterns.

```
CREATE INDEX idx_order_customerid ON "order"(customerid);
CREATE INDEX idx_od_ordernumber ON orderdetail(ordernumber);
CREATE INDEX idx_product_categoryname ON product(categoryname);
CREATE INDEX idx_product_lowstock_partial
  ON product(currentstock)
  WHERE currentstock < reorderpoint;
```

Query performance was evaluated with PostgreSQL execution plans using `EXPLAIN (ANALYZE, BUFFERS)`.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT DATE_TRUNC('month', o.orderdate) AS month,
       SUM(od.quantity * od.unitprice) AS total_sales
FROM "order" o
INNER JOIN orderdetail od
  ON o.ordernumber = od.ordernumber
GROUP BY month
ORDER BY month;
```

Access control was modeled through job-based roles and GRANT statements.

```
CREATE ROLE order_clerk LOGIN PASSWORD '...';
CREATE ROLE inventory_manager LOGIN PASSWORD '...';

GRANT SELECT ON public.menuitem, public.product, public.category TO order_clerk;
GRANT INSERT ON public."order", public.orderdetail TO order_clerk;
GRANT SELECT, INSERT, UPDATE ON public.product, public.productsupply
  TO inventory_manager;
```

Transaction handling was demonstrated with atomic order entry and stock deduction using BEGIN/COMMIT. The original work also discussed savepoint-style partial rollback, write-ahead logging, and point-in-time recovery as recovery concepts.

SQL vs NoSQL / Architecture Reflection

The project considered how document databases such as MongoDB could complement the relational system. PostgreSQL is well suited for orders, payments, inventory, role-based access, and integrity-sensitive reporting. A document model could be useful for flexible product catalog content, café menu presentation, marketing content, customer preferences, or other semi-structured data that changes frequently.

This comparison demonstrates architectural thinking: choosing data stores based on data shape, integrity requirements, query patterns, and operational needs rather than treating SQL and NoSQL as interchangeable technologies.