

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

Introduction to the AdventureWorks Cycles Case Study

AdventureWorks Cycles is a well-established bicycle manufacturer that has been a key player in the cycling industry for many years. The company specializes in producing and selling a diverse range of bicycles, including road, mountain, and hybrid models, alongside a variety of cycling accessories such as helmets and gear. Renowned for its dedication to quality and innovation, AdventureWorks Cycles is embarking on a mission to enhance its operations and customer experience.

To achieve this, the company is developing a state-of-the-art online marketplace where customers can browse, customize, and purchase bicycles and accessories from various manufacturers. This transformation requires a reliable and scalable database system to efficiently manage essential business operations such as product management, customer interactions, order processing, and feedback collection.

The focus of this case study is to design, implement, and optimize the database system to support AdventureWorks Cycles' operational and analytical needs. Key objectives include:

- Defining relationships among entities like customers, products, manufacturers, orders, and feedback.
- Supporting frequent queries for product browsing, order tracking, and feedback analysis.
- Maintaining data consistency and reliability through robust transactional operations.
- Improving query performance with well-designed secondary indexes for critical operations.

By deploying this database solution, AdventureWorks Cycles aims to streamline its processes, enhance customer satisfaction, and maintain a competitive advantage in the market. This report provides a comprehensive overview of the database design, implementation, and optimization strategies undertaken to meet these goals.

Entity Relationship Diagram Summary

The Entity Relationship Diagram (ERD) for AdventureWorks Cycles outlines the essential entities, their attributes, and the relationships that underpin the company's business operations. It includes the **Manufacturer** entity, which records details of product producers, and the **Category** entity, which organizes products into groups such as road bikes and accessories. Product data is stored in the **Product** entity, containing attributes like name, description, price,

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

and a reference to the manufacturer. The many-to-many relationship between products and categories is represented using the **ProductCategory** junction table. Customer details, including names and email addresses, are managed in the **Customer** entity, while purchase transactions are tracked in the **Order** entity, which links orders to customers. Specific details of the products in each order, such as quantities and subtotals, are captured in the **OrderDetail** entity. Additionally, the **Feedback** entity stores customer reviews and ratings, connecting customers and products. Relationships between entities are maintained through primary and foreign keys, ensuring data integrity and enabling efficient queries to support the company's operational needs. The **cardinality** between entities ensures accurate representation of relationships. For example, one customer can place multiple orders, while each order is linked to one customer, creating a one-to-many relationship. Similarly, the many-to-many relationship between products and categories is modeled through the junction table, allowing a product to belong to multiple categories and a category to include multiple products (Fig. 1.).

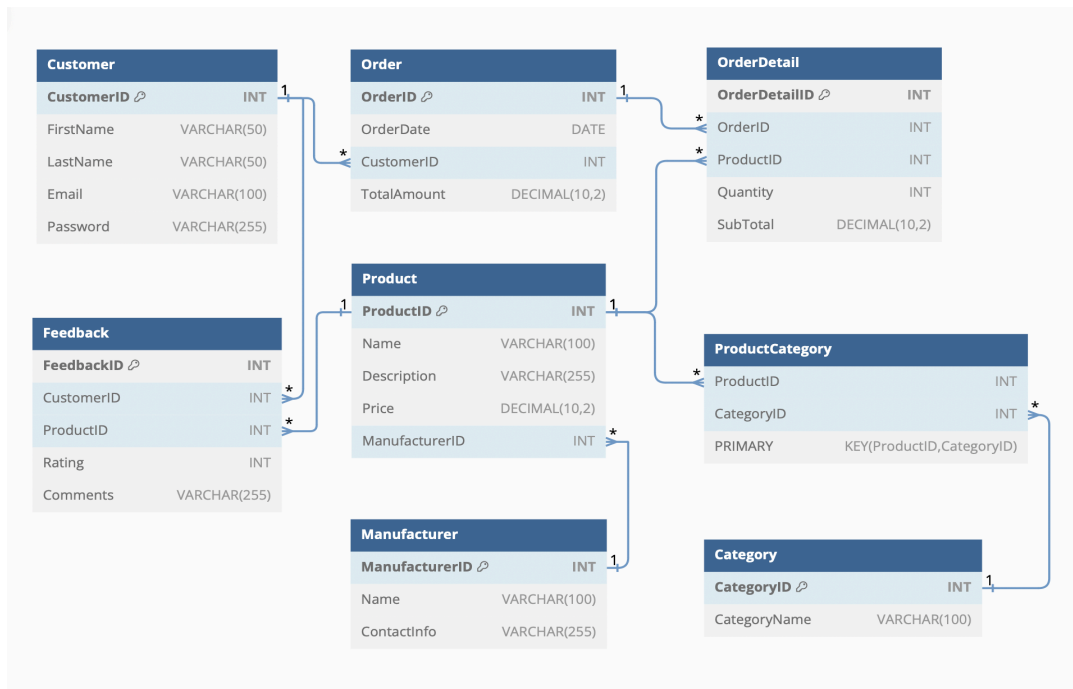


Fig. 1. The Entity Relationship Diagram (ERD) for AdventureWorks Cycles

Database design

The database and tables for AdventureWorks Cycles were created to represent the entities and their relationships as defined in the ERD. The SQL code includes CREATE TABLE statements for each entity, specifying the attributes and data types (Fig. 2.). Each table is designed to store specific data, such as customers, products,

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

orders, and feedback. Primary keys were defined to uniquely identify records in each table, ensuring data consistency and enabling relationships between entities.

Foreign key constraints were added to maintain referential integrity across the database. These constraints link related tables, such as connecting CustomerID in the Order table to the Customer table, or ManufacturerID in the Product table to the Manufacturer table. Foreign keys ensure that referenced data exists in the parent table, preventing orphan records and maintaining the logical consistency of the database.

Check constraints were implemented to enforce data validity by restricting the range or format of values allowed in certain columns. For example, the Price column in the Product table includes a constraint to ensure prices are non-negative, and the Rating column in the Feedback table enforces a range between 1 and 5. These constraints improve data quality by preventing invalid entries and ensuring adherence to business rules (Fig. 3. a. and b.) .

A trigger was created on the OrderDetail table to automatically calculate the SubTotal whenever a new record is inserted or updated (Fig. 4.). This ensures that the SubTotal is dynamically derived based on the product price and quantity, maintaining consistency and reducing the risk of manual calculation errors.

The database was filled with sample data using Insert statements (Fig. 5.).

```
-- Step 1: Create the database
CREATE DATABASE AdventureWorksCycles;

-- Use the created database
USE AdventureWorksCycles;
```

Fig.2. The creation of the database

```
-- Step 8: Create the OrderDetail table
CREATE TABLE OrderDetail (
    OrderDetailID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incrementing primary key
    OrderID INT NOT NULL,
    ProductID INT NOT NULL,
    Quantity INT CHECK (Quantity > 0), -- Enforce positive quantities
    SubTotal DECIMAL(10, 2) CHECK (SubTotal >= 0), -- Enforce non-negative subtotal
    FOREIGN KEY (OrderID) REFERENCES [Order](OrderID), -- Add referential integrity constraint
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID) -- Add referential integrity constraint
);

-- Step 9: Create the Feedback table
CREATE TABLE Feedback (
    FeedbackID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incrementing primary key
    CustomerID INT NOT NULL,
    ProductID INT NOT NULL,
    Rating INT CHECK (Rating BETWEEN 1 AND 5), -- Enforce rating range of 1 to 5
    Comments NVARCHAR(255),
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID), -- Add referential integrity constraint
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID) -- Add referential integrity constraint
);
```

Fig.3. a. Creation of the database tables and constraints

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

```
-- Step 2: Create the Manufacturer table
CREATE TABLE Manufacturer (
    ManufacturerID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incrementing primary key
    Name NVARCHAR(100) NOT NULL,
    ContactInfo NVARCHAR(255)
);

-- Step 3: Create the Category table
CREATE TABLE Category (
    CategoryID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incrementing primary key
    CategoryName NVARCHAR(100) NOT NULL
);

-- Step 4: Create the Product table
CREATE TABLE Product (
    ProductID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incrementing primary key
    Name NVARCHAR(100) NOT NULL,
    Description NVARCHAR(255),
    Price DECIMAL(10, 2) CHECK (Price >= 0), -- Enforce non-negative prices
    ManufacturerID INT NOT NULL,
    FOREIGN KEY (ManufacturerID) REFERENCES Manufacturer(ManufacturerID) -- Add referential integrity constraint
);

-- Step 5: Create the ProductCategory table (junction table for many-to-many relationship)
CREATE TABLE ProductCategory (
    ProductID INT NOT NULL,
    CategoryID INT NOT NULL,
    PRIMARY KEY (ProductID, CategoryID), -- Composite primary key
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID), -- Add referential integrity constraint
    FOREIGN KEY (CategoryID) REFERENCES Category(CategoryID) -- Add referential integrity constraint
);

-- Step 6: Create the Customer table
CREATE TABLE Customer (
    CustomerID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incrementing primary key
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    Email NVARCHAR(100) UNIQUE NOT NULL, -- Enforce unique email
    Password NVARCHAR(255) NOT NULL
);

-- Step 7: Create the Order table
CREATE TABLE [Order] (
    OrderID INT IDENTITY(1,1) PRIMARY KEY, -- Auto-incrementing primary key
    OrderDate DATE NOT NULL,
    CustomerID INT NOT NULL,
    TotalAmount DECIMAL(10, 2) CHECK (TotalAmount >= 0), -- Enforce non-negative totals
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) -- Add referential integrity constraint
);
```

Fig.3. b. Creation of the database tables and constraints

```
CREATE TRIGGER trg_CalculateSubTotal
ON OrderDetail
AFTER INSERT, UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    -- Update the SubTotal for the affected rows
    UPDATE OrderDetail
    SET SubTotal = OD.Quantity * P.Price
    FROM OrderDetail OD
    JOIN Product P ON OD.ProductID = P.ProductID
    WHERE OD.OrderDetailID IN (
        SELECT OrderDetailID FROM inserted
    );
END;
```

Fig. 4. Creating trigger for automatic calculation of Subtotal

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

```
-- Step 1: Populate Manufacturer Table
INSERT INTO Manufacturer (Name, ContactInfo)
VALUES
('BikePro', 'contact@bikepro.com'),
('CycleWorld', 'info@cycleworld.com'),
('UrbanCycles', 'support@urbancycles.com'),
('TrailMasters', 'info@trailmasters.com');

-- Step 2: Populate Category Table
INSERT INTO Category (CategoryName)
VALUES
('Road Bikes'),
('Mountain Bikes'),
('Hybrid Bikes'),
('Accessories');

-- Step 3: Populate Product Table
INSERT INTO Product (Name, Description, Price, ManufacturerID)
VALUES
('Roadster 200', 'High-speed road bike', 1200.00, 1), -- Manufacturer: BikePro
('Mountain Climber', 'Durable mountain bike', 1500.00, 2), -- Manufacturer: CycleWorld
('Urban Commuter', 'Lightweight hybrid bike', 800.00, 3), -- Manufacturer: UrbanCycles
('Trail Buddy Helmet', 'Protective cycling helmet', 100.00, 4); -- Manufacturer: TrailMasters

-- Step 4: Populate ProductCategory Table
INSERT INTO ProductCategory (ProductID, CategoryID)
VALUES
(1, 1), -- Roadster 200 in Road Bikes
(2, 2), -- Mountain Climber in Mountain Bikes
(3, 3), -- Urban Commuter in Hybrid Bikes
(4, 4); -- Trail Buddy Helmet in Accessories

-- Step 5: Populate Customer Table
INSERT INTO Customer (FirstName, LastName, Email, Password)
VALUES
('John', 'Doe', 'john.doe@example.com', 'password123'),
('Jane', 'Smith', 'jane.smith@example.com', 'securepassword'),
('Alex', 'Johnson', 'alex.johnson@example.com', 'mypassword'),
('Emily', 'Davis', 'emily.davis@example.com', 'emilypass');

-- Step 6: Populate Order Table
INSERT INTO [Order] (OrderDate, CustomerID, TotalAmount)
VALUES
('2025-01-08', 1, 1200.00), -- Customer: John
('2025-01-09', 2, 1500.00), -- Customer: Jane
('2025-01-10', 3, 800.00), -- Customer: Alex
('2025-01-11', 4, 100.00); -- Customer: Emily
```

Fig. 5. Inserting data into the database

Transactions

The transactional scenario for AdventureWorks Cycles is designed to demonstrate the use of commit and rollback operations in a database system. This scenario involves multiple SQL statements to simulate a real-world operation where a customer places an order, and the system records the order, its details, and the total amount. The transaction ensures that all operations are completed successfully as a single unit or entirely reverted if an error occurs, maintaining data integrity.

The process begins by inserting a new record into the Order table for a specific customer. The OrderID generated from this insertion is used to associate multiple

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

products in the OrderDetail table. Each order detail includes the product's quantity and calculated subtotal based on the product price. Finally, the total amount for the order is updated in the Order table by summing the subtotals from the OrderDetail records.

The transaction is enclosed within a TRY...CATCH block to handle errors gracefully. If all operations execute successfully, the transaction is committed, saving the changes to the database. However, if any statement fails—such as inserting a non-existent ProductID or violating constraints—the transaction is rolled back to ensure no partial data is saved (Fig. 6. a. and b.).

The design of this transactional scenario aligns with the principles of atomicity, ensuring that all steps succeed or none are applied. This prevents inconsistent or incomplete data in the database. It also incorporates robust error handling, with detailed error messages for debugging. The scenario reflects a real-world use case, such as processing customer orders, where multiple interdependent operations must be executed reliably. By incorporating both commit and rollback conditions, this scenario demonstrates the database system's ability to handle complex transactions while preserving data integrity and consistency.

```
BEGIN TRANSACTION; -- Start the transaction

-- Step 1: Insert a new order
INSERT INTO [Order] (OrderDate, CustomerID, TotalAmount)
VALUES ('2025-01-15', 2, 0.00); -- Customer: Jane

-- Get the generated OrderID
DECLARE @NewOrderID INT = SCOPE_IDENTITY();

-- Step 2: Insert valid order details
INSERT INTO OrderDetail (OrderID, ProductID, Quantity, SubTotal)
VALUES (@NewOrderID, 1, 1, 1 * (SELECT Price FROM Product WHERE ProductID = 1)); -- Roadster 200

INSERT INTO OrderDetail (OrderID, ProductID, Quantity, SubTotal)
VALUES (@NewOrderID, 2, 2, 2 * (SELECT Price FROM Product WHERE ProductID = 2)); -- Mountain Climber

-- Step 3: Update the total amount in the Order table
UPDATE [Order]
SET TotalAmount = (SELECT SUM(SubTotal) FROM OrderDetail WHERE OrderID = @NewOrderID)
WHERE OrderID = @NewOrderID;

-- Commit the transaction
COMMIT TRANSACTION;

PRINT 'Transaction committed successfully.';

END TRY

-- Rollback the transaction in case of an error
ROLLBACK TRANSACTION;

PRINT 'Transaction rolled back due to an error.';
PRINT ERROR_MESSAGE(); -- Display the error message

END CATCH;

-- Check the newly created order
SELECT * FROM [Order] WHERE OrderDate = '2025-01-15';

-- Check the order details for the new order
SELECT * FROM OrderDetail WHERE OrderID = (
    SELECT OrderID FROM [Order] WHERE OrderDate = '2025-01-15'
);
```

Fig. 6. a. Transaction with commit scenario

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

```
BEGIN TRANSACTION; -- Start the transaction

-- Step 1: Insert a new order
INSERT INTO [Order] (OrderDate, CustomerID, TotalAmount)
VALUES ('2025-01-16', 3, 0.00); -- Customer: Alex

-- Get the generated OrderID
DECLARE @NewOrderID INT = SCOPE_IDENTITY();

-- Step 2: Insert valid order detail
INSERT INTO OrderDetail (OrderID, ProductID, Quantity, SubTotal)
VALUES (@NewOrderID, 1, 1, 1 * (SELECT Price FROM Product WHERE ProductID = 1)); -- Roadster 200

-- Step 3: Intentionally cause an error (invalid ProductID)
INSERT INTO OrderDetail (OrderID, ProductID, Quantity, SubTotal)
VALUES (@NewOrderID, 9999, 1, 1 * (SELECT Price FROM Product WHERE ProductID = 9999)); -- Invalid ProductID

-- Step 4: Update the total amount in the Order table
UPDATE [Order]
SET TotalAmount = (SELECT SUM(SubTotal) FROM OrderDetail WHERE OrderID = @NewOrderID)
WHERE OrderID = @NewOrderID;

-- Commit the transaction
COMMIT TRANSACTION;

PRINT 'Transaction committed successfully.';
END TRY

-- BEGIN CATCH
-- Rollback the transaction in case of an error
ROLLBACK TRANSACTION;

PRINT 'Transaction rolled back due to an error.';
END CATCH;

-- Check if the order was created
SELECT * FROM [Order] WHERE OrderDate = '2025-01-16';

-- Check order details for the failed transaction
SELECT * FROM OrderDetail WHERE OrderID = (
    SELECT OrderID FROM [Order] WHERE OrderDate = '2025-01-16'
);
```

Fig. 6. b. Transaction with rollback scenario

Performance Tuning Enhancement

To enhance query performance for AdventureWorks Cycles, the most critical transactions and frequently executed queries were identified, and appropriate secondary indexes were proposed. These queries include retrieving products by category, manufacturer, and price range, viewing customer orders and their details, analyzing feedback for specific products, and generating reports on recent orders. Secondary indexes were strategically added to optimize these queries (Fig. 7.). For example, indexing the `Price` and `ManufacturerID` columns in the `Product` table improves product browsing efficiency, while an index on `CategoryID` in the `ProductCategory` table accelerates lookups for products by category. Similarly, indexing `CustomerID` in the `Order` table enhances the performance of queries fetching a customer's order history, and indexing `OrderDate` supports efficient range-based queries for reporting. The `OrderDetail` table benefits from an index on `OrderID` to speed up retrieval of detailed information for specific orders. Lastly, indexing `ProductID` in the `Feedback` table ensures fast analysis of customer reviews for particular products. These indexes were chosen based on their ability to

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

reduce query execution time, minimize resource usage, and improve overall database responsiveness for common operations.

```
CREATE INDEX idx_Product_Price_Manufacturer
ON Product (Price, ManufacturerID);

CREATE INDEX idx_ProductCategory_CategoryID
ON ProductCategory (CategoryID);

CREATE INDEX idx_Order_CustomerID
ON [Order] (CustomerID);

CREATE INDEX idx_Order_OrderDate
ON [Order] (OrderDate);

CREATE INDEX idx_OrderDetail_OrderID
ON OrderDetail (OrderID);

CREATE INDEX idx_Feedback_ProductID
ON Feedback (ProductID);
```

Fig. 7. Defining secondary indexes

Databases and Application Development, Assignment, Zsolt Gyongyosi, 2227989, Applied Computing

References

1. Connolly, T., & Begg, C. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson Education.
2. Microsoft. (2023). *SQL Server Documentation*. Retrieved from <https://learn.microsoft.com/sql>