Byte Wizards
Group 17

Project Phase 1 - Database Initial Study,

Project Phase 2 - Database Design

&

Project Phase 3 - Physical Design

(CMPG311)

Group Members:

[41218787 – I. Senekal] – (Group Leader)

[41763882 – R. van Heerden]

[44214987 – J. Cloete]

[41550226 – S. Buys]

[42320755 – R. Swart]

[37955039 – D. Niebuhr]

Submission date: 19/05/2024

TABLE OF CONTENTS

Project Phase 1 - Database Initial Study

Members of the group

 [41218787 – I. Senekal] – (Group Leader)	 [41763882 – R. van Heerden]
 [44214987 – J. Cloete]	 [41550226 – S. Buys]
 [42320755 – R. Swart]	 [37955039 – D. Niebuhr]

Analyse company situation

Company Objectives(mission of the environment)

The Dunder Mifflin Paper Company mission is to provide high-quality paper products and stationery to digital small-business clients and tech-savvy individuals. Their slogan reflects this mission: “Limitless paper in a paperless world”

The Dunder Mifflin Paper Company objectives to reach their mission involve:

- Obtaining profitability and stability
- Ensuring effective communication between departments
- Establishing a positive work environment through fair wages and compensation
- Offer high-quality products at the correct time place and to the correct person
- Reducing carbon footprint by using less paper and moving to a more digitised system
- Ensure data integrity and security of customer and employee information
- Monitor stock levels, orders and supplier information

Company Operations(general operating environment)

The Dunder Mifflin Paper Company bears a unique workforce, each with a set of rare skills. In analysing the company situation, it is clear that the skills of these employees are not fully utilised to contribute to the goals and objectives of the company. The main reason behind this is not a lack of motivation or productivity, but rather inadequate internal business operations that have not been properly defined, maintained and implemented.

In the following section we will take a look at the current company operations

Employee Management

Dunder Mifflin’s current time tracking system relies on manual punch cards, which employees use to clock their time when starting their daily shift and clocking out after the days’ work. Employee salaries are strictly calculated based on the employee work hours. Managers will often spend a significant amount of time manually reconciling timecards. As a result, the processing and adjustments may delay payrolls and even lead to inaccuracies in employee pay.

The branch has no centralised system to track attendance, performance evaluations and leave requests. Human Resources (HR) is compelled to maintain paper files and folders for each employee. Paper profiles and employee information is handwritten, thus when a mistake or change needs to be made, a large part of the folders content is discarded, and everything needs to be filled in and filed from the start. It is mandatory to fill out a leave template on paper and issue it to the head of the employees’ department.

After this the file is then submitted to the office of the applicable HR member. In the past this process has led to various problems. One of which is the instance where there are significant delays in leave requests, impacting the morale and productivity of the workforce. Salary increases and bonuses are determined by the results of the yearly employee evaluation. There are no set criteria for the evaluation process and each department is responsible for their own evaluation criteria.There have also been instances noted by management where evaluation of performance reviews was defective because of the inability to access various employee data effectively.

Current File System

The current file system relies heavily on paper-based filing cabinets scattered throughout the office. There are no specific set out locations or assigned cabinets to file different documents like sale orders, invoices or contracts.Sale orders are placed by customers telephonically or in person when a customer visits the branch and engages with a sales agent. The customer is then quoted and upon payment a paper handwritten invoice is then issued to the customer. These important documents are stored unsystematically and results in a hit-and-miss for each attempt to find a specific file in various cabinets.

There have also been reports of invalid employment agreements whereby an employee has requested a new copy of their work contract, just to be notified their contract could not be found at the moment.

Fortunately, the Paper Company has at least one technological aspect: a company shared drive. This is for sharing and storing files digitally between different departments. Upon closer inspection it is obvious that the state of their digital workspace is the same as their physical office workspace. There are no folders for sorting relevant files and files are not using any standard naming conventions.

Additional operational deficiency with their digital shared drive, is that there are no limitations to access to any of the files or directories.

Warehouse Management

As we have established thus far, the company's main operations are centred around physical file management. This means that currently in a scenario where a customer places a new order, the order information must be physically picked up by the fulfilment team at the warehouse (or faxed to them) before the products can be shipped to the customer.

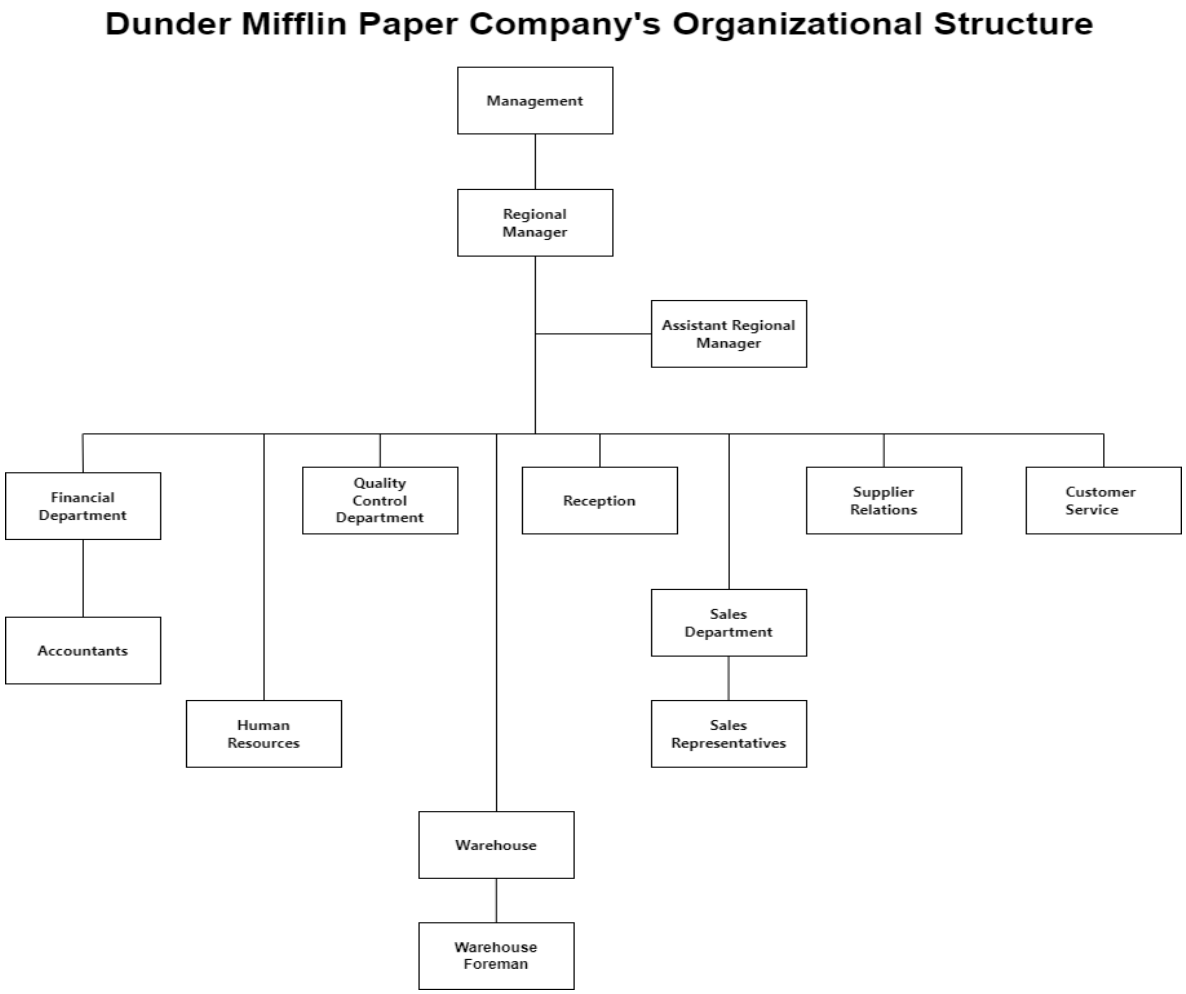
There are no communication systems currently in place (except communication over a telephone) to transfer any new orders, cancellations or updates between the sales department and the warehouse.Additionally, the warehouse relies on handwritten logs and spreadsheets that make up the current inventory tracking system. As a result, there is no real-time visibility of inventory levels

There is no track record of suppliers and deliveries. Management will often change suppliers based on the most competitive prices, often leading to new delivery dates and shipment information. The uninformed warehouse staff will regularly complain about the inconsistency, resulting in confusion and frustration as to which inventory items are supplied and delivered by which supplier.

Company Organisational Structure

In this section, we will analyse all the departments within Dunder Mifflin Paper Company's organisational structure and evaluate their roles, responsibilities, and interconnectedness to gain a comprehensive understanding of the company's operation.

Figure 1: Dunder Mifflin Paper Company, Inc. Organisational Structure



Regional Manager

Within our organisation the Regional Manager has access and control over all the departments below him shown above in Figure 1. Likewise he also enables communication and collaboration between these departments to ensure a prosperous future for the company.

Assistant Regional Manager

Serves as an assistant to the Regional Manager, where he might compile financial documents and reports of the company's current status, for the Regional Manager to assess the quality of work, and to review the results against the goals that have been set.

Financial Department

The financial department’s tasks include things such as: budgeting, payroll processing, invoicing, as well as finalising contracts the business acquires. The budgeting involves a structured process of planning, allocating, and monitoring financial resources to specific departments to achieve the organisation's objectives and goals.

Quality Control Department

The goal of this department is to meet the predetermined standards that have been set before production began. These standards are applied to the products and services that the company produces over a certain time frame. As with all things these standards need constant improvement and maintenance which could help the business to be more resource efficient. Some forms of improving your business would be to ask your suppliers and retailers to complete a comprehensive survey about their experience doing business with you, or by investigating the methods other similar businesses use.

Warehouse

The warehouse serves as the storage facility for raw materials as well as the end products that have been produced. They receive orders which will then be efficiently organised inside the warehouses and the file systems' inventory to make it easier to package outgoing shipments, orders and deliveries to the necessary branches and retailers.

Reception/Secretary (Secretarial Department)

The secretary of this department handles all the deliveries that are not meant for the warehouse, furthermore they also greet and inform the clients and employees that enter the main building. They answer queries customers have about general information concerning the business such as business hours and topics concerning the goods and services that are being provided.

Sales Department

The purpose of the Sales Department is to communicate with potential and current clients to negotiate and then finalise contracts in order to deliver on the products and services the company provides to retailers and individual customers.

By maintaining a collaborative relationship between other companies we can recommend and or suggest new deliverables to them which will show that we are capable, willing and able to adjust and quickly adapt to new market and consumer trends. They also work hand in hand with the Supplier Relations Department.

Supplier Relations

This department facilitates clear and concise communication and agreements between businesses that lead to a type mutual or Business-to-business (B2B) relationship between entities. By maintaining a positive relationship between one another, they can provide assistance to each other by collaborating and facing challenges together.

Customer Service

When a business prioritises customer service it leads to contracts being established as well as supplier and retailer loyalty which in turn creates a stable income within the business. Thus Inside a business environment these clients will take the top-priority.

Define problems and constraints

PROBLEMS

Employee Problems:

- **Manual clock in system:** the system is vulnerable to human errors and abuse resulting in inaccurate representation of work hours. Employees may forget to punch in or out.
- **Inaccurate Break Tracking:** There is no system in place to track breaks accurately. Making it impossible to distinguish between the scheduled work hours and the actual hours worked.

File System Problems:

- **Outdated filing and employee management system:** The branch uses a cabinet filing system and consists of various departments that work independently, leading to multiple copies, human error and a chance of data anomalies.
- **Redundancy:** Duplicate files are created due to human error which leads to data redundancy in the filing system.
- **Outdated files**: Multiple files can exist in different locations. Once the original file is found, it does not contain the updated information.
- **Supplier annoyance:** The outdated filing system causes delays in supplying correct files to suppliers. The supplier may receive incorrect files, resulting in wrong or insufficient product orders.
- **Shipping department annoyance:** Shipping works on a tight schedule due to the old filing system, it is time-consuming to get the correct files. Incorrect files may be provided to the shipping department, resulting in loss of time and money for the branch.

Warehouse Problems:

- **Warehouse management system:** The stock is maintained in the paper filing system which is outdated. There are redundant files leading to incorrect stock count or ordering of new stock.
- **Inventory Control:** Inventory misuse results in overstocking of specific goods and stockouts of high-demand products. This system is prone to human error and misrepresentation of the stock warehouse.

General Problems:

- **Security:** Due to the outdated filing system and shared drive(current computer system) any employee can get access to files they don't have authorization to.
- **Reports:** Due to the filing system, it takes a long time to generate different types of reports. This leads to late or incorrect ordering of stock, poor management of employees, and wrong information of stock count.

CONSTRAINTS

- **Financial constraints:** The branch has a limited budget for expansion and automation of their system.
- **Infrastructure limitations:** The branch faces infrastructure limitations such as outdated office facilities, inadequate technology infrastructure, and unreliable utilities.
- **Local regulations and compliance:** The branch needs to comply with specific local regulations and ordinances that differ from those in other locations where the company operates, adding complexity and potential costs to its operations.
- **Suppliers:** There can be multiple suppliers to the same product.
- **Stuck in old ways:** It can be difficult to convince the manager and employees that an automated system is the best course of action, because what they have has been working for them.
- **Dependency on corporate decisions:** The manager of the branch may agree, but they still need confirmation from the head company.

Database system specification

Define Objectives to solve problems:

Employee Objectives:

- **Time tracking:** Implementing a system that accurately logs the employee's hours, breaks, and overtime. By using an electronic check-in and check-out system. The system will have a report function for time spent on tasks. This gives the manager a way to compare the time they spent with the time they planned.

- **Reward system:** This must be made specifically for each department. Since the average work and time spent on tasks are different, this will serve as motivation for employees to work harder.

- **Improved digital platform**: Creating a GUI that shows each employee's time log, tasks, and performance metrics. This will help the manager maintain a healthy work environment and maintain good employee records.

File System - Digitized System:

- **Centralization of all data:** To create a central database system that stores all the company data and reports. This will allow for easy access to data and information.

- **Security:** Ensuring the data that is stored is secure, and distributed correctly. Also implementing access controls based on employee status, to prevent unauthorised access.

Warehouse Management:

- **Digitised communication between supplier and business:** The system will connect the business with the suppliers. It will produce automated notifications, when stock is running low a certain person will receive a message to order more, this is for niche items where bulk items will be ordered automatically.

- **Optimization of management:** To optimise management, by implementing inventory tracking via barcodes. Set up automated recordings with the help of barcodes to keep track of all the items in our stock.

Information the company requires from the database:

- An up to date list of employees with their personal information
- Which employee has access to what part of the system and their associated role
- Managers must be able to see a list of all employee work hours as well as their overtime and break logs
- Employee salaries and bonuses, that are automatically calculated based on their stored rates and work hours
- Placed sales orders with their associated details, such as customer details, delivery details and requested items and quantities.
- The status of an order (completed, not started, in progress, cancelled)
- List of available stock
- Inventory levels of each stock item stored
- Suppliers and their contact details
- What stock which supplier fulfils
- Transaction history with each supplier

Scope

Our proposed system's scope includes, designing and implementing an integrated database management system to meet Dunder Mifflin Paper Company's operational needs.

Which will include things such as:

- Automating time tracking and attendance management.
- Monitor and rewarding employee performance
- Centralising HR management processes.
- Digitising document management and file storage.
- Upgrading warehouse operations with digital inventory management.
- Enhancing security measures and access controls.
- Extensive reporting including financial and stock reports
- Maintain entities

Boundaries

Hardware: At the moment Dunder Mifflin Paper company has an outdated hardware system. Which would make it difficult to use new software. The database system therefore cannot be too complex. Hardware has to be compatible with the new system implemented.

Software: Currently they have systems for sales and accounting, which has to be integrated with the new system. Their software system is up-to-date therefore Windows 11 will be needed by users using their own devices. In addition database software(Oracle) will be needed on all computers within the business.

Financial: Dunder Mifflin Paper company offered us a limited budget to complete the database system. Since they are a mid-size company they do not have the funds for high-end services and expensive servers to store data

Time: Dunder Mifflin gave us a time constraint with this project. The complete system and database must be implemented and operational within 6 months.

Project Phase 2 - Database Design

Introduction

This document presents a comprehensive conceptual and logical design of a database system made for the needs of Dunder Mifflin. In the document the following are goals that were achieved: defining the business rules, Entity-Relationship Diagram (ERD), and normalisation of the database to ensure efficiency and reliability.

The Business Rules are derived from the company's operations, these rules will outline relationships and entities within Dunder Mifflin, such as those concerning employee management, the file system, and warehouse management. After creating the ERD and undergoing normalisation, we refined our business rules to align with the identified entities and relationships.

The ERD represents all the logical structure of the database, depicting entities, attributes, primary keys, foreign keys, and various relationships. It also illustrates examples of mandatory and optional relationships, strong and weak entities, and composite entities.

In the logical design we use normalisation to eliminate redundancy, partial dependencies and improve the data integrity. By following the steps of database design we aim to provide a robust foundation for managing the company's operations effectively.

The following few changes were adding **Payrolls** and **Bank**. We needed this information to simplify employee and supplier payments. After normalisation a few extra entities were also identified.

Since Dunder Mifflin workforce is not fully experienced with a database system to ensure no attributes are added which mean the same thing for example in the entity **Logged_Time** the action for leaving work could be added as “logged out” or “check out”, each entity with this problem is resolved by adding a Type entity. In the database the following of these kinds of entities were added: **Bank_Name**, **Account_Type**, **Employee_Role** and **Action_Type**(for example mentioned above)

Conceptual Design

Business Rules

Employee Management Business Rules based on Organisation Operations

These business rules concern the section on the organisation operation that were identified before the ERD was drawn up and Normalizations was completed . The following business rule will show

- The relationship employees have with departments, their logged times(clocking in and out),
- The file system and the relationship that it has with customers orders.
- The warehouse relationship with suppliers and the relationship between the warehouse management with customers and with the orders they placed.

EMPLOYEE, LOGGED TIMES & DEPARTMENTS

- An Employee can record none to many Logged Times
- One Logged Time is record by one Employee
- One Employee is associated with one Department.
- One Department has many Employees.

FILE SYSTEM

- Orders are placed by a Customer telephonically
- A Customer can place one to many Orders telephonically
- Orders are placed by a Customers in person
- One Customer can place one to many Orders in person

WAREHOUSE MANAGEMENT

- Each Customer can place one to many Orders.
- One Order belongs to one Customer.
- One Customer can receive one to Many Products
- Many Products can be shipped to one Customer.
- None to one Shipment is carried out for one Order
- One Order is fulfilled by one Shipment
- One Supplier supplies one to many Products
- One to many Products are supplied by one Supplier
- Records are kept of Suppliers transactions
    - One Supplier is involved in many Transactions
    - One Transaction has one Supplier involved

Refined Business Rules based on ERD(Normalisation)

After the Logical design steps and ERD the following entities and additional entities were identified with the relationships between them.

PAYROLL

- One Employee receives one Payroll
- One Payroll is received by one Employee

LOGGED TIMES

- An Employee can record none to many Logged Times
- Logged Times are recorded by one Employee
- One Action Type can have none to many Logged Times
- One Logged Time has one Action Type

DEPARTMENT

- One Employee has one Employee Role
- Many to None Employees can have one Employee Role
- One Employee Role belongs to one Department
- One Department has one to many Employee Roles

BANK

- One Employee has one Bank Detail
- One Bank Detail belongs to one Employee
- One Bank Detail has one Bank Name
- One Bank Name could corresponds to none to many Bank Details
- One Bank Details has one Account Type
- One Account Type could corresponds to none to many Bank Details
- One Bank Details can belongs to none to one Supplier
- One Supplier has one Bank Detail

CUSTOMERS

- One customer has one to many Orders
- One Order belong to one Customer

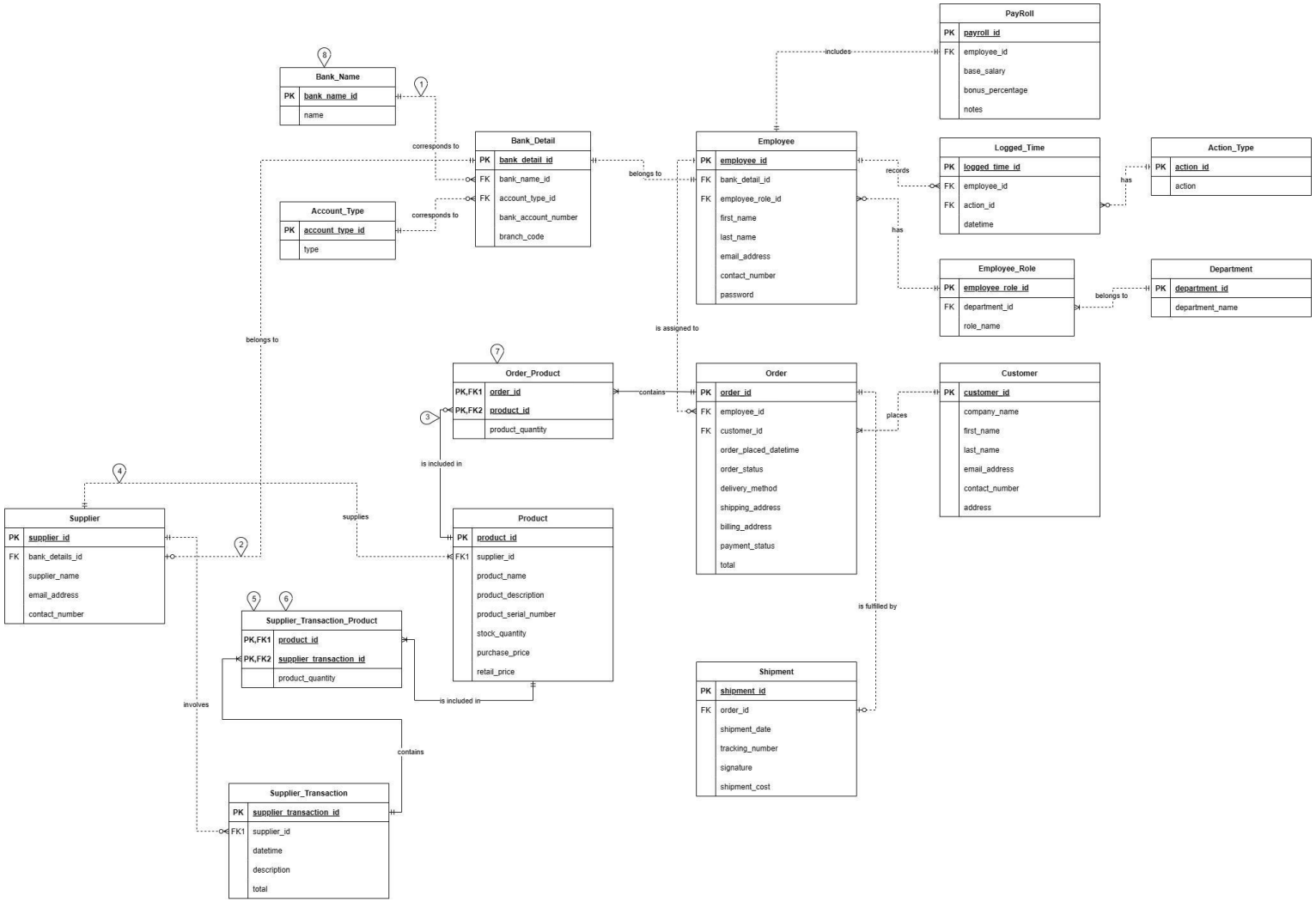
ORDERS

- One Order has one to many Order Products
- One Order Products has one Order
- One Order Products includes none to many Product
- One Product is include in one Order Product
- One Order is fulfilled by none to one Shipment
- One Shipment has one Order
- One Order is assigned to one Employee
- One Employee has none to many Orders

SUPPLIERS

- One Supplier supplies one to many Products
- One Product has one Supplier
- One Supplier has none to many Supplier Transactions
- One Supplier Transactions has one Supplier
- One Product is included in one to many Supplier Transactions Product
- One Supplier Transactions Product includes one Product
- One Supplier Transaction contains one to many Supplier Transactions Product
- One Supplier Transactions Product is contained in one Supplier Transaction

ER Diagram



To more clearly view the ER-Diagram of ThePaperCompany, click this link → https://tinyurl.com/mr2cv3nw

Meaning of Numbered Icons

1. Example of a **mandatory relationship** between Bank_Name and Bank_Detail.
    - An Employee and Supplier cannot give their Bank_Details without supplying a Bank_Name.

2. Example of an **optional relationship** between Supplier and Bank_Detail.
    - An entity occurrence in the Supplier table does not necessarily require the existence of a corresponding entity occurrence in the Bank_Detail table.

3. Example of a **strong relationship**
    - The Order_Product entity primary key is composed of product_id and order_id. Therefore, a strong relationship exists between Product and Order_Product because product_id (the primary key of the parent entity) is a primary key component in the Order_Product entity. In other words, the Order_Product primary key did inherit a primary key component from the Product entity.

4. Example of a **weak relationship.**
    - The Product primary key(product_id) did not inherit a primary key component from the Supplier entity. In this case, a weak relationship exists between Product and Supplier because supplier_id(the primary key of the parent entity) is only a foreign key in the Product entity.

5. Example of a **composite / bridge entity**.
    - The composite entity's (Supplier_Transaction_Product) primary key consists of the primary keys of the entities that it connects(product_id and supplier_transaction_id)

6. Order_Product is an example of a **weak entity.**
    - An entity that displays existence dependence and inherits the primary key of its parent entity. For example, Order_Product requires the existence of a Product

7. Bank_Name is an example of a **strong entity.**
    - A strong entity is one that resides in the "1" side of all its relationships—that is, an entity that does not have a mandatory attribute that is a foreign key to another table. Bank_Name does not receive a foreign key from another entity and it resides on the "1" side.

Entity clarification (Rikus Swart)

The purpose of this section is to provide some clarity regarding the structure and representation of certain entities in our database.

- Order_Product: We have two separate entities. Firstly, Order is responsible for tracking the details of the order. Each order has at least one product that exists in the Product entity. There arises an obvious problem that when an order has multiple products, it is easy to fall into the trap of logging an redundant amount of records representing duplicate order details for each product that is associated with an order. To combat this, we created an entity called Order_Product. Each order is logged with its associated product. This way we can associate multiple products with each order and allocate them with an added quantity attribute.

- Supplier_Transaction_Product: Our database design accommodates the process of keeping track of the history of transactions that happen between a supplier and the organisation. Each transaction with a supplier can include multiple products, resulting in the same redundancy problem previously faced. To combat this we implemented an entity, Supplier_Transaction_Product that clearly indicates the association between each transaction and corresponding products that were involved.

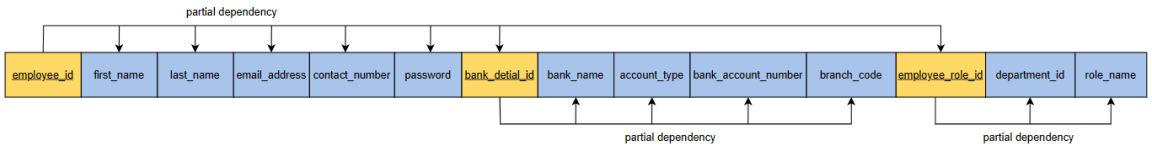
- Logged_Times: Each time an employee takes a break, clocks in, clocks out or any other predetermined action, a logged time is recorded for precise employee tracking and management. Each logged time is associated with an action that indicates what the record entry is for.

- Bank_Name, Account_Type, Action_Type: These are some examples of entities that exist to ensure data integrity remains our primary focus. This also makes the database scalable in terms of not having fixed column values, but rather having the option to later add to the entity and enabling the user to select predetermined values that exist in these entities, minimising human errors.

Logical design

Employee, Bank_Detail & Employee_Role

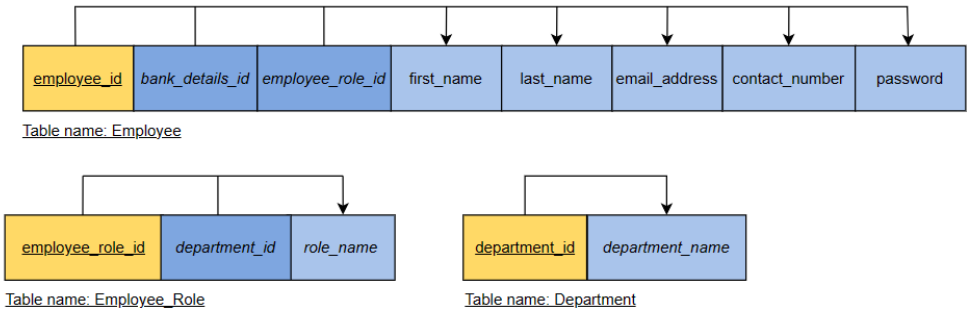
First Normal Form:



1NF:
(employee_id, bank_detail_id, employee_role_id, first_name, last_name, email_address, contack_number, password, bank_name, account_type, bank_account_number, branch_code, department_id, role_name)
Partial Dependencies:

- (employee_id = *bank_detail_id, employee_id,* first_name, last_name, email_address, contact_number, password)
- (bank_detail_id = bank_name, account_type, bank_account_number, branch_code)
- (employee_role_id, *department_id,* role_name)

Third Normal Form:



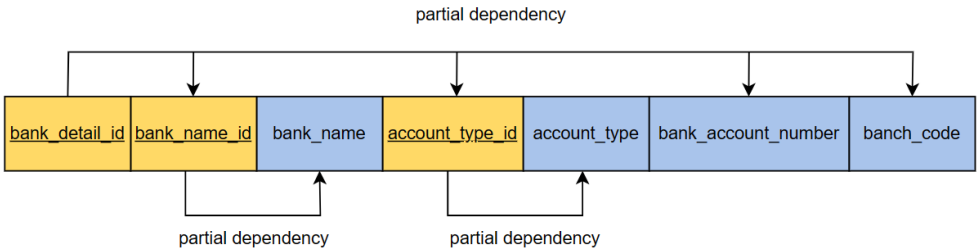
Employee(employee_id(PK), *bank_details_id*(FK1), *employee_role_id*(FK2), first_name, last_name, email_address, contact_number, password)

Employee_Role(employee_role_id(PK), *department_id*(FK1), role_name)

Department(department_id(PK), department_name)

Bank_Detail & Bank_Name, Acoount_Type

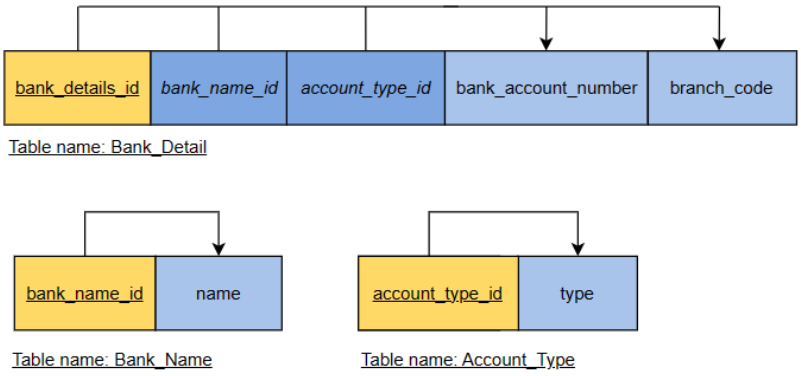
First Normal Form:



1NF:
(bank_detail_id, bank_name_id, account_type_id, bank_name, account_type, bank_account_number, branch_code)
Partial Dependencies:

- (bank_detail_id = *bank_name_id, account_type_id,* bank_account_number, branch_code)
- (bank_name_id = name)
- (acount_type_id_ = type)

Third Normal Form:



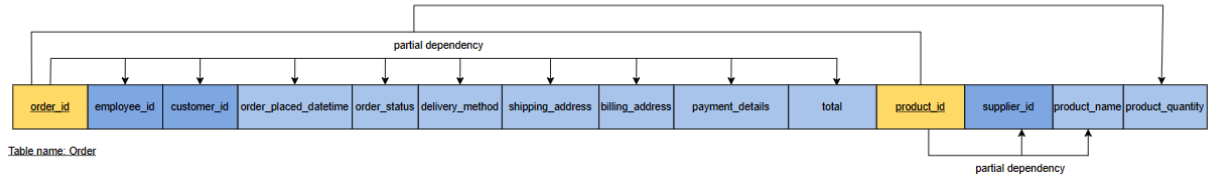
Bank_Details(bank_details_id(PK), *bank_name_id*(FK1), *account_type_id*(FK2), bank_account_number, branch_code)

Bank_Name(bank_name_id(PK), name)

Account_Type(account_type_id(PK), type)

Order, Product & Ordered_Product

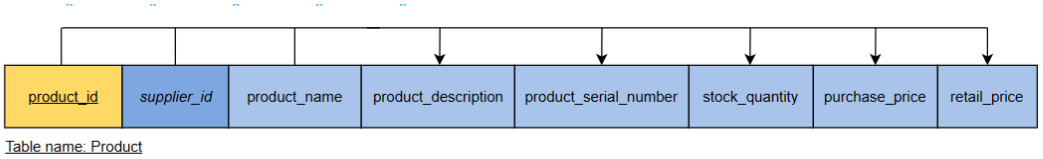
First Normal Form:

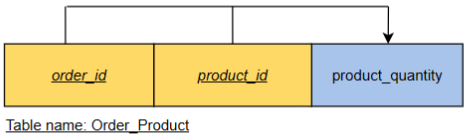


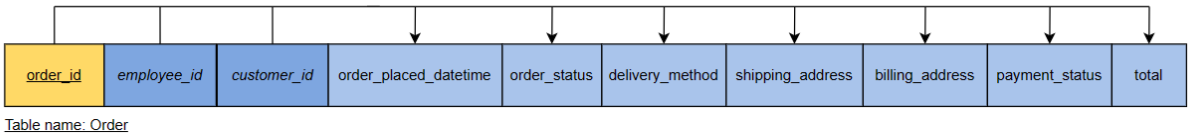
1NF:
(order_id, product_id, *employee_id, customer_id,* order_placed_datetime, order_status, delivery_method, shipping_address, billing_address, payment_details, total, product_name, product_quantity)
Partial Dependencies:

- (order_id = *employee_id, customer_id,* order_placed_datetime, order_status, delivery_method, shipping_address, billing_address, payment_details, total)
- (order_id, product_id = product_quantity)
- (person_id_ = *supplier_id*, product_name)

Third Normal Form:







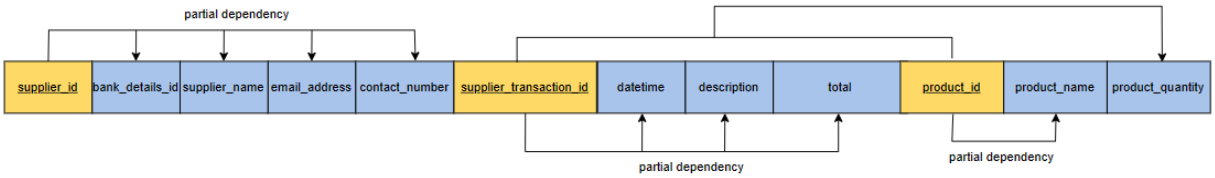
Product(product_id(PK), *supplier_id*(FK1), product_name, product_description, product_serial_number, stock_quantity, purchase_price, retail_price)

Ordered_Products(*order_id(PK)(FK1), product_id(PK)(FK2)*, product_quantity)

Order(order_id(PK), *employee_id*(FK1), *customer_id*(FK2),order_placed_datetime, order_status, delivery_method, shipping_address, billing_address, payment_status, total)

Supplier, Supplier_Transactions & Supplier_Transaction_Product

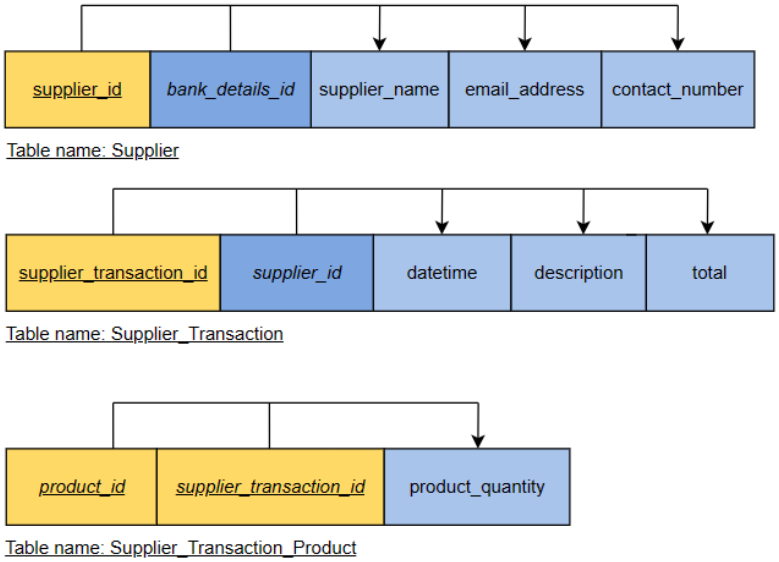
First Normal Form:



1NF:
(supplier_id, supplier_transaction_id,product_id, *bank_details_id, supplier_name,* email_address, contact_number, datetime, description, total, product_name, product_quantity)
Partial Dependencies:

- (supplier_id = *bank_details_id, supplier_name,* email_address, contact_number)
- (supplier_transaction_id = datetime, description, total)
- (product_id_ = product_name)
- (supplier_transaction_id, product_id_ = product_quantity)

Third Normal Form:



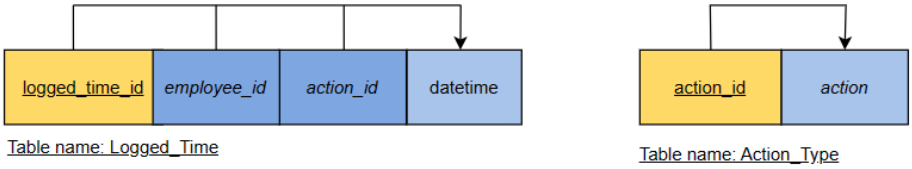
Supplier(supplier_id(PK), *bank_details_id*(FK1), supplier_name, email_address, contact_number)

Supplier_Transaction(supplier_transaction_id(PK), *supplier_id*(FK1), date_time, description, total)

Supplier_Transaction_Product(*product_id(PK)(FK1) supplier_transaction_id(PK)(FK2***)**, product_quantity)

Logged_Time & Action Type

Third Normal Form:



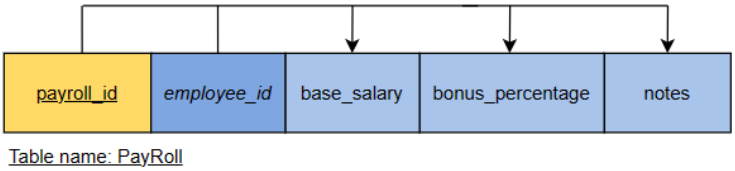
Logged_Time(logged_time_id(PK), *employee_id*(FK1), *action_id*(FK2), date_time)

Action_Type(action_id(PK), action)

The following are already in 3NF

Payroll

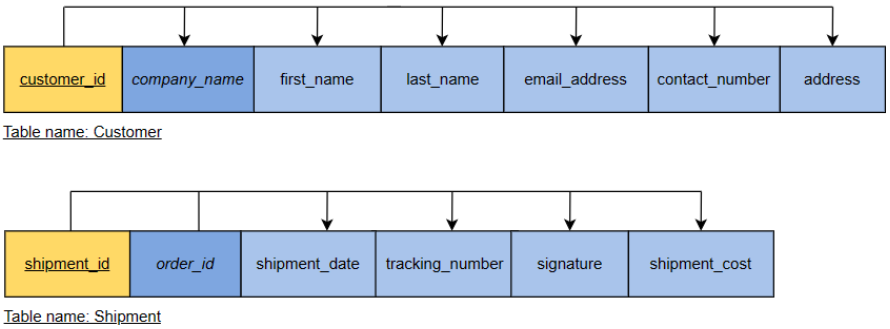
Third Normal Form:



PayRoll(payroll_id(PK), *employee_id*(FK1), base_salary, bonus_percentage, notes)

Customer & Shipment

Third Normal Form:



Customer(customer_id(PK), company_name, first_name, last_name, email_address, contact_number, address)

Shipment(shipment_id(PK),*order_id*(FK1), shipment_date, tracking_number, signature, shipment_cost))

Summary of Final Logical Design after Normalisation

Employee(employee_id(PK), *bank_details_id*(FK1), *employee_role_id*(FK2), first_name, last_name, email_address, contact_number, password)

Employee_Role(employee_role_id(PK), *department_id*(FK), role_name)

Department(department_id(PK), department_name)

Logged_Time(logged_time_id(PK), *employee_id*(FK1), *action_id*(FK2), date_time)

Action_Type(action_id(PK), action)

PayRoll(payroll_id(PK),*employee_id*(FK), base_salary, bonus_percentage, notes)

Bank_Details(bank_details_id(PK), *bank_name_id*(FK1), *account_type_id*(FK2), bank_account_number, branch_code)

Bank_Name(bank_name_id(PK), bank_name)

Account_Type(account_type_id(PK), type)

Supplier(supplier_id(PK), *bank_details_id*(FK), supplier_name, email_address, contact_number)

Supplier_Transaction(supplier_transaction_id(PK), *supplier_id*(FK1), date_time, description, total)

Supplier_Transaction_Product(*product_id(PK)(FK1), supplier_transaction_id(PK)(FK2)***)**, product_quantity)

Product(product_id(PK), *supplier_id*(FK1), product_name, product_description, product_serial_number, stock_quantity, purchase_price, retail_price)

Ordered_Products(*order_id(PK)(FK1), product_id(PK)(FK2)*, product_quantity)

Order(order_id(PK), *employee_id*(FK1), *customer_id*(FK2),order_placed_datetime, order_status, delivery_method, shipping_address, billing_address, payment_status, total)

Customer(customer_id(PK), company_name, first_name, last_name, email_address, contact_number, address)

Shipment(shipment_id(PK),*order_id*(FK1), shipment_date, tracking_number, signature, shipment_cost))

Project Phase 3 - Physical Design

Introduction

Our journey with Dunder Mifflin started off with the aim to improve the performance, management and accessibility of the company and its information. Taking on an already thriving company posed some unique challenges.

From the outside, this unique paper company is perceived to entail a simple organisational structure, while maintaining an effective workflow and productive workforce. Upon closer inspection, we quickly realised this is not the case. The company consists of various departments, each with their own set of roles and associated employees. Furthermore, it became evident the company must maintain multiple connections with everything from suppliers right through to customers, while simultaneously maintaining up to date information of inventory, finances and shipments.

Our challenge was set and we started by evaluating the company in its current situation. We laid out the company's goals and objectives and mapped out the organisational structure and all of its intricacies. We analysed this information and highlighted areas where we can realistically improve the company's current systems and also add beneficial value to certain areas.

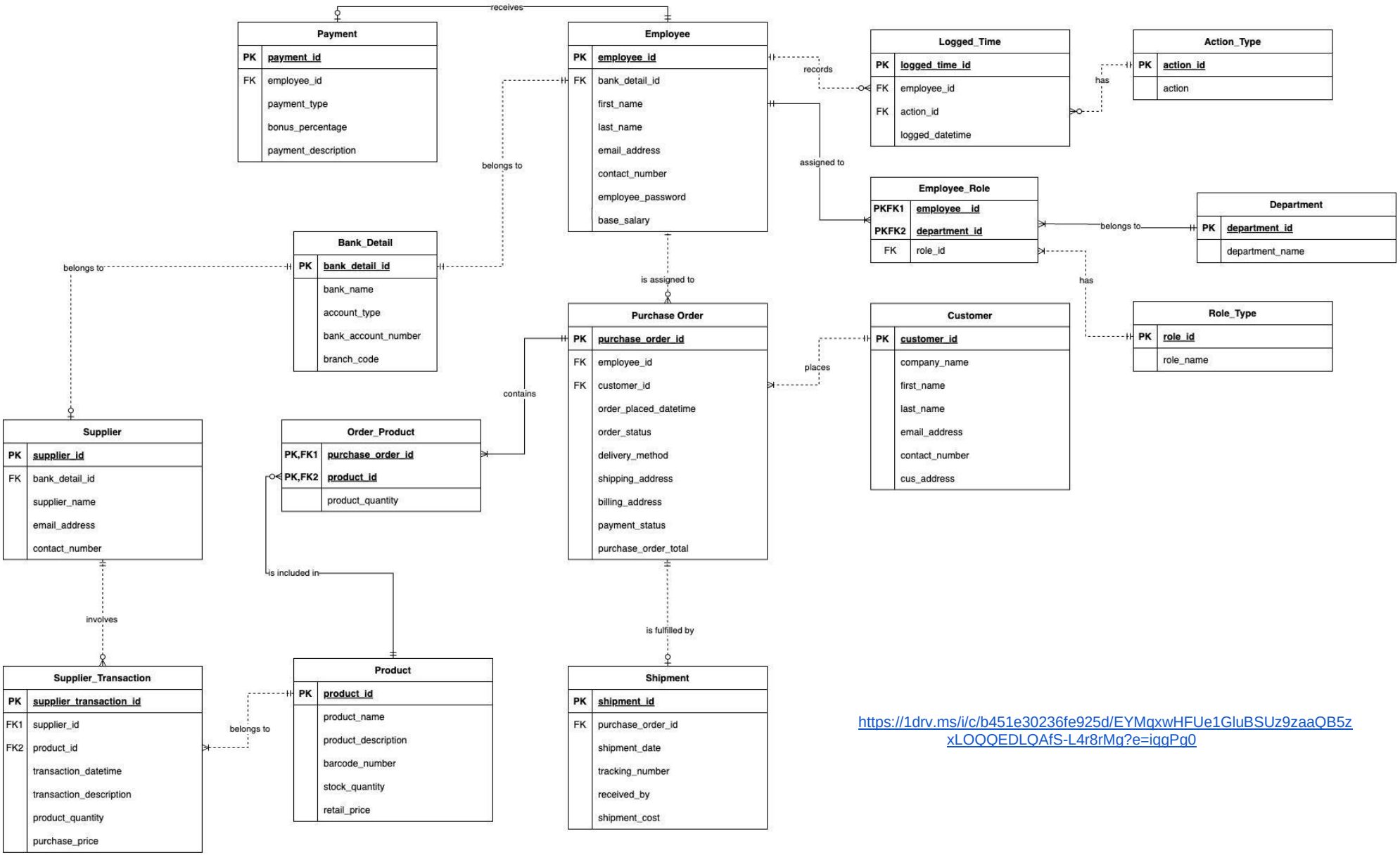
The initial planning we have drawn out was never set in stone, and this allowed us to quickly adapt our approach when facing a challenge. Through changing our entry points to various situations multiple times, we have crafted a system that enables Dunder Mifflin to excel beyond expectations.

This document is composed of different sections with regards to the complete planning specifically of the database section. Each section is carefully laid out with detailed planning. It is important to note that the following sections are each sequential, following a logical order of creating a truly fascinating system.

Prior to the sections is our finalised and adapted ERD. The ERD gives a broad overview of the database and its tables, as well as the relationships among those tables. As previously mentioned, our dynamic approach allowed us to change the planning to suit certain needs. After closer inspection we have removed certain redundant tables with regards to the banking details of the employee and the supplier. Furthermore, we have updated the table name from payroll to payment, to eliminate any confusion to the actual functionality of this table. Another major change was the restructuring of the relationship between suppliers, supplier transactions and products. We have minimised the complexity of these tables by ensuring each only contains fields that are appropriate to them. Lastly, we have updated or removed certain unnecessary fields that do not add beneficial value to the system.

Each of the sections following the ERD is broken down into a detailed explanation, to explain our approach of creating the database.

Updated ERD based on updated/added Requirements



https://1drv.ms/i/c/b451e30236fe925d/EYMqxwHFUe1GluBSUz9zaaQB5zxLOQQEDLQAfS-L4r8rMg?e=iggPg0

Database Objects

Drop Tables

The reason for dropping the tables is to ensure consistency throughout the creation of the database. It ensures that issues related to table structures of previous tables do not cause problems. The order in which tables are dropped is dependent on foreign key constraints. Therefore the common rule of thumb is to drop child tables before parent tables to maintain data integrity and to avoid errors due to foreign key constraints.

```
-- Drop Tables --
DROP TABLE order_product;

DROP TABLE employee_role;

DROP TABLE department;

DROP TABLE role_type;

DROP TABLE logged_time;

DROP TABLE action_type;

DROP TABLE shipment;

DROP TABLE purchase_order;

DROP TABLE customer;

DROP TABLE supplier_transaction;

DROP TABLE product;

DROP TABLE supplier;

DROP TABLE payment;

DROP TABLE employee;

DROP TABLE bank_detail;
```

Drop Sequences

Existing sequences may cause conflicts if new definitions are provided, which is why they are dropped before tables are created. By dropping them it ensures consistency and it prevents unintended data from affecting the new sequences and tables. Unlike tables, sequences do not have to be dropped in a specific order. Further in the document sequences will be explained in more detail.

```
-- Drop Sequences --
DROP SEQUENCE action_id_seq;
DROP SEQUENCE department_id_seq;
DROP SEQUENCE role_id_seq;
DROP SEQUENCE bank_detail_id_seq;
DROP SEQUENCE product_id_seq;
DROP SEQUENCE customer_id_seq;
DROP SEQUENCE employee_id_seq;
DROP SEQUENCE logged_time_id_seq;
DROP SEQUENCE purchase_order_id_seq;
DROP SEQUENCE shipment_id_seq;
DROP SEQUENCE supplier_id_seq;
DROP SEQUENCE supplier_transaction_id_seq;
DROP SEQUENCE payment_id_seq;
```

Tables/Entities Creation & Constraints

The following tables were created based on the entities identified in Dunder Mifflin's Paper Companys' business rules. The relationships and constraints will also be indicated and will be explained more in detail for each table. It should also be noted that the tables identified are based on the updated ERD.

It can be noted throughout that since the keyword PRIMARY KEY is used the NOT NULL constraint is not necessary. However it is indicated to enforce this constraint in all tables.

STRONG(No Foreign Keys) ENTITIES

The following tables show strong entities which do not have any foreign keys as attributes. This means all entities are existence independent(they do not depend on another entity to exist).

Action_Type

The DDL query creates a table called ACTION_TYPE with two attributes.

- **action_id** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **action** has a data type of VARCHAR2 which only allows for 25 character values.

Both attributes have a NOT NULL constraint.

```
CREATE TABLE action_type (
    action_id   NUMBER NOT NULL PRIMARY KEY, --NOT NULL contraint
    action      VARCHAR2(25) NOT NULL
);
```

Department

The DDL query creates a table called DEPARTMENT with two attributes.

- **department_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **department_name:** has a data type of VARCHAR2 which only allows for 25 character values.

Both attributes have a NOT NULL constraint.

```
CREATE TABLE department (
    department_id     NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    department_name   VARCHAR(25) NOT NULL
);
```

Role_Type

The DDL query creates a table called ROLE_TYPE with two attributes.

- **role_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **role_name:** has a data type of VARCHAR2 which only allows for 15 character values.

Both attributes have a NOT NULL constraint.

```
CREATE TABLE role_type (
    role_id     NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    role_name   VARCHAR2(15) NOT NULL
);
```

Bank_Detail

The DDL query creates a table called BANK_DETAIL with five attributes.

- **bank_detail_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **bank_name:** has a data type of VARCHAR2 which only allows for 25 character values.
- **account_type:** has a data type of VARCHAR2 which only allows for 25 character values.
- **bank_account_number:** has a data type of VARCHAR2 which only allows for 20 character values.
- **branch_code:** has a data type of VARCHAR2 which only allows for 6 character values.

All attributes have a NOT NULL constraint.

```
CREATE TABLE bank_detail (
    bank_detail_id        NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    bank_name             VARCHAR2(25) NOT NULL,
    account_type          VARCHAR2(25) NOT NULL,
    bank_account_number   VARCHAR2(20) NOT NULL,
    branch_code           VARCHAR2(6) NOT NULL
);
```

Product

The DDL query creates a table called PRODUCT with six attributes.

- **product_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **product_name:** has a data type of VARCHAR2 which only allows for 125 character values.
- **product_description:** has a data type of VARCHAR2 which only allows for 125 character values.
- **barcode_number:** has a data type of VARCHAR2 which only allows for 25 character values, the attribute also has the UNIQUE constraint
- **stock_quantity:** has a data type of NUMBER
- **retail_price:** has a data type of NUMBER
- All attributes have a NOT NULL constraint except **stock_quantity** and **retail_price**

A CHECK constraint is applied to stock_quantity to ensure that the column never contains a negative value or only contains a value greater or equal to 0. The CHECK constraint is also applied to retail_price to ensure the column never contains a negative value.

```
CREATE TABLE product (
    product_id            NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    product_name          VARCHAR2(125) NOT NULL,
    product_description   VARCHAR2(125) NOT NULL,
    barcode_number        VARCHAR2(25) UNIQUE NOT NULL, --UNIQUE contraint
    stock_quantity        NUMBER,
    retail_price          NUMBER,
    CHECK ( stock_quantity >= 0 ), --Check Constaint
    CHECK ( retail_price > 0 ) --Check Constaint
);
```

Customer

The DDL query creates a table called CUSTOMER with seven attributes.

- **customer_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **company_name:** has a data type of VARCHAR2 which only allows for 20 character values.
- **first_name:** has a data type of VARCHAR2 which only allows for 20 character values.
- **last_name:** has a data type of VARCHAR2 which only allows for 20 character values, the attribute also has the UNIQUE constraint
- **email_address:** has a data type of VARCHAR2 which only allows for 35 character values.
- **contact_number:** has a data type of VARCHAR2 which only allows for 10 character values.
- **cus_address:** has a data type of VARCHAR2 which only allows for 100 character values.

All attributes have a NOT NULL constraint except **company_name** since a customer can either be an individual or a company

```
CREATE TABLE customer (
    customer_id      NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    company_name     VARCHAR2(20),
    first_name       VARCHAR2(20) NOT NULL,
    last_name        VARCHAR2(20) NOT NULL,
    email_address    VARCHAR2(35) NOT NULL,
    contact_number   VARCHAR2(10) NOT NULL,
    cus_address          VARCHAR2(100) NOT NULL
);
```

STRONG (with foreign keys) ENTITIES

The following tables show strong entities which do have foreign keys as attributes.

Employee

The DDL query creates a table called EMPLOYEE with eight attributes.

- **employee_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **bank_detail_id:** has a data type of NUMBER.
- **first_name:** has a data type of VARCHAR2 which only allows for 20 character values.
- **last_name:** has a data type of VARCHAR2 which only allows for 20 character values, the attribute also has the UNIQUE constraint
- **email_address:** has a data type of VARCHAR2 which only allows for 35 character values.
- **contact_number:** has a data type of VARCHAR2 which only allows for 10 character values.
- **employee_password:** has a data type of VARCHAR2 which only allows for 15 character values.
- **base_salary:** has a data type of NUMBER.

A CHECK constraint is applied to **base_salary** to ensure that the column never contains a negative value or only contains a value greater or equal to 0. Since salaries can never be negative

The last few lines indicate a FOREIGN KEY constraint on **bank_detail_id.** This means that the value in the bank_detail_id column of EMPLOYEE must also exist as a primary key in the BANK_DETAIL table.

The ON DELETE CASCADE ensures that if a employee is deleted the corresponding record in the bank_detail table is also deleted

```
CREATE TABLE employee (
    employee_id         NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    bank_detail_id      NUMBER,
    first_name          VARCHAR2(20) NOT NULL,
    last_name           VARCHAR2(20) NOT NULL,
    email_address       VARCHAR2(35) NOT NULL,
    contact_number      VARCHAR2(10) NOT NULL,
    employee_password   VARCHAR2(15) NOT NULL,
    base_salary        NUMBER NOT NULL,
    CHECK ( base_salary >= 0 ),  --Check Constaint
    FOREIGN KEY ( bank_detail_id )
        REFERENCES bank_detail ( bank_detail_id )
            ON DELETE CASCADE
);
```

Logged_Time

The DDL query creates a table called LOGGED_TIME with four attributes.

- **logged_time_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **employee_id:** has a data type of NUMBER
- **action_id:** has a data type of NUMBER..
- **logged_datetime:** has a data type of TIMESTAMP which only allows for storing the date and the time.

All attributes have a NOT NULL constraint.

The last few lines indicate a FOREIGN KEY constraint on **employee_id** and **action_id.** This means that:

- The value in the employee_id column of LOGGED_TIME must also exist as a primary key in the EMPLOYEE table.
- The value in the action_id column of LOGGED_TIME must also exist as a primary key in the ACTION_TYPE table.

```
CREATE TABLE logged_time (
    logged_time_id    NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    employee_id       NUMBER NOT NULL,
    action_id         NUMBER  NOT NULL,
    logged_datetime   TIMESTAMP  NOT NULL,
    FOREIGN KEY ( employee_id )
        REFERENCES employee ( employee_id ),
    FOREIGN KEY ( action_id )
        REFERENCES action_type ( action_id )
);
```

Purchase_Order

The DDL query creates a table called PURCHASE_ORDER with ten attributes.

- **purchase_order_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **employee_id:** has a data type of NUMBER
- **customer_id:** has a data type of NUMBER..
- **order_place_datetime:** has a data type of TIMESTAMP.
- **order_status:** has a data type of VARCHAR2 which only allows for 15 character values.
- **delivery_method:** has a data type of VARCHAR2 which only allows for 20 character values.
- **shipping_address:** has a data type of VARCHAR2 which only allows for 55 character values
- **billing_address:** has a data type of VARCHAR2 which only allows for 70 character values.
- **paymen_ status:** has a data type of VARCHAR2 which only allows for 35 character values.
- **purchase_order_total:** has a data type of NUMBER.

All attributes have a NOT NULL constraint except for purchase_order_number which has CHECK constraints.

- A CHECK constraint is applied to **purchase_order_number** to ensure that the column never contains a negative value. Since the total purchase cannot equal zero

The last few lines indicate a FOREIGN KEY constraint on **employee_id** and **customer_id.** This means that:

- The value in the employee_id column of PURCHASE_ORDER must also exist as a primary key in the EMPLOYEE table.
- The value in the customer_id column of PURCHASE_ORDER must also exist as a primary key in the CUSTOMER table.

```
CREATE TABLE purchase_order (
    purchase_order_id       NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    employee_id             NUMBER NOT NULL,
    customer_id             NUMBER NOT NULL,
    order_placed_datetime   TIMESTAMP NOT NULL,
    order_status            VARCHAR2(15) NOT NULL,
    delivery_method         VARCHAR2(20) NOT NULL,
    shipping_address        VARCHAR2(55) NOT NULL,
    billing_address         VARCHAR2(70) NOT NULL,
    payment_status          VARCHAR2(35) NOT NULL,
    purchase_order_total    NUMBER,
    CHECK ( purchase_order_total > 0 ), --Check Constaint
    FOREIGN KEY ( employee_id )
        REFERENCES employee ( employee_id ),
    FOREIGN KEY ( customer_id )
        REFERENCES customer ( customer_id )
);
```

Shipment

The DDL query creates a table called SHIPMENT with six attributes.

- **shipment_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **purchase_order_id:** has a data type of NUMBER..
- **shipment_date:** has a data type of DATE.
- **tracking_number:** has a data type of VARCHAR2 which only allows for 15 character values.
- **received_by:** has a data type of VARCHAR2 which only allows for 40 character values.
- **shipment_cost:** has a data type of NUMBER.

All attributes have a NOT NULL constraint except for shipment_cost which has CHECK constraints.

- A CHECK constraint is applied to **shipment_cost** to ensure that the column never contains a negative value.

The last few lines indicate a FOREIGN KEY constraint on **purchase_order_id.** This means that:

- The value in the purchase_order_id column of SHIPMENT must also exist as a primary key in the PURCHASE_ORDER table.

```
CREATE TABLE shipment (
    shipment_id         NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    purchase_order_id   NUMBER NOT NULL,
    shipment_date       DATE NOT NULL,
    tracking_number     VARCHAR2(15) NOT NULL,
    received_by         VARCHAR2(40) NOT NULL,
    shipment_cost       NUMBER,
    CHECK ( shipment_cost >= 0 ), --Check Constaint
    FOREIGN KEY ( purchase_order_id )
        REFERENCES purchase_order ( purchase_order_id )
);
```

Supplier

The DDL query creates a table called SUPPLIER with five attributes.

- **supplier_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **bank_detail_id:** has a data type of NUMBER.
- **supplier_name:** has a data type of VARCHAR2 which only allows for 35 character values.
- **email_address:** has a data type of VARCHAR2 which only allows for 25 character values.
- **contact_number:** has a data type of VARCHAR2 which only allows for 10 character values.

All attributes have a NOT NULL constraint

The last few lines indicate a FOREIGN KEY constraint on **bank_detail_id.** This means that the value in the bank_detail_id column of SUPPLIER must also exist as a primary key in the BANK_DETAIL table.

The ON DELETE CASCADE ensures that if a supplier is deleted the corresponding record in the bank_detail table is also deleted

```
CREATE TABLE supplier (
    supplier_id      NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    bank_detail_id   NUMBER NOT NULL,
    supplier_name    VARCHAR2(35) NOT NULL,
    email_address    VARCHAR2(25) NOT NULL,
    contact_number   VARCHAR2(10) NOT NULL,
    FOREIGN KEY ( bank_detail_id )
        REFERENCES bank_detail ( bank_detail_id )
            ON DELETE CASCADE
);
```

Supplier_Transaction

The DDL query creates a table called SUPPLIER_TRANSACTION with seven attributes.

- **supplier_transaction_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **supplier_id:** has a data type of NUMBER.
- **product_id:** has a data type of NUMBER.
- **transaction_datetime:** has a data type of TIMESTAMP.
- **transaction_description:** has a data type of VARCHAR2 which only allows for 125 character values.
- **product_quantity:** has a data type of NUMBER.
- **purchase_price:** has a data type of NUMBER.

All attributes have a NOT NULL constraint except **product_quantity** and **purchase_price.**

- A CHECK constraint is applied to product_quantity to ensure that the column never contains a negative value or only contains a value greater or equal to.
- The CHECK constraint is also applied to purchase_price to ensure the column never contains a negative value.

The last few lines indicate a FOREIGN KEY constraint on **supplier_id** and **product_id.** This means that:

- The value in the supplier_id column of SUPPLIER_TRANSACTION must also exist as a primary key in the SUPPLIER table.
- The value in the product_id column of SUPPLIER_TRANSACTION must also exist as a primary key in the PRODUCT table.

```
CREATE TABLE supplier_transaction (
    supplier_transaction_id   NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    supplier_id               NUMBER NOT NULL,
    product_id                NUMBER NOT NULL,
    transaction_datetime      TIMESTAMP NOT NULL,
    transaction_description   VARCHAR2(125) NOT NULL,
    product_quantity          NUMBER,
    purchase_price            NUMBER,
    CHECK ( purchase_price >= 0 ), --Check Constaint
    CHECK ( product_quantity > 0 ), --Check Constaint
    FOREIGN KEY ( supplier_id )
        REFERENCES supplier ( supplier_id ),
    FOREIGN KEY ( product_id )
        REFERENCES product ( product_id )
);
```

Payment

The DDL query creates a table called PAYMENT with five attributes.

- **payment_id:** has a data type of NUMBER which is also the primary key of the table which is why the PRIMARY KEY keyword is used.
- **employee_id:** has a data type of NUMBER.
- **payment_type:** has a data type of VARCHAR2 which only allows for 25 character values, the attribute also has the UNIQUE constraint
- **bonus_percentage:** has a data type of NUMBER
- **payment_description:** has a data type of VARCHAR2 which only allows for 125 character values.

All attributes have a NOT NULL constraint except bonus_percentage

- A CHECK constraint is applied to **bonus_percentage** to ensure that the column never contains a negative value or only contains a value greater or equal to 0.

The last few lines indicate a FOREIGN KEY constraint on **employee_id.** This means that the value in the employee_id column of PAYMENT must also exist as a primary key in the EMPLOYEE table.

```
CREATE TABLE payment (
    payment_id          NUMBER NOT NULL PRIMARY KEY, --NOT NULL constraint
    employee_id         NUMBER NOT NULL,
    payment_type       VARCHAR2(25) NOT NULL,
    bonus_percentage    NUMBER NOT NULL,
    payment_description   VARCHAR2(125) NOT NULL,
    CHECK ( bonus_percentage >= 0 ), --Check Constaint
    FOREIGN KEY ( employee_id )
        REFERENCES employee ( employee_id )
);
```

WEAK/BRIDGE_ENTITIES

The following tables represent weak entities. This means that the entity is existence-dependent(the entity depends on a strong entity for its existence). Some foreign keys form part of this entity's primary key.

Order_Product

The DDL query creates a table called ORDER_PRODUCT with three attributes.

- **purchase_order_id:** has a data type of NUMBER.
- **product_id:** has a data type of NUMBER.
- **product_quantity:** has a data type of NUMBER.

All attributes have a NOT NULL constraint except product_quantity

- A CHECK constraint is applied to **product_quantity** to ensure that the column never contains a negative value.

The two lines under the check constraint defines a Composite Primary Key containing (**purchase_order_id** + **product_id**)

The last few lines indicate a FOREIGN KEY constraint on **purchase_order_id** and **product_id.** This means that

- The value in the purchase_order_id column of ORDER_PRODUCT must also exist as a primary key in the PURCHASE_ORDER table.
- The value in the product_id column of ORDER_PRODUCT must also exist as a primary key in the PRODUCT table.

The ON DELETE CASCADE ensures that if an order_product is deleted the corresponding record in the purchase_order table is also deleted.

```
CREATE TABLE order_product (
    purchase_order_id   NUMBER NOT NULL, --NOT NULL constraint
    product_id          NUMBER NOT NULL, --NOT NULL constraint
    product_quantity    NUMBER,
    CHECK ( product_quantity > 0 ), --Check Constaint
    PRIMARY KEY  ( purchase_order_id,
                   product_id ),
    FOREIGN KEY ( purchase_order_id )
        REFERENCES purchase_order ( purchase_order_id )
            ON DELETE CASCADE,
    FOREIGN KEY ( product_id )
        REFERENCES product ( product_id )
);
```

Employee_Role

The DDL query creates a table called EMPLOYEE with three attributes.

- **employee_id:** has a data type of NUMBER.
- **department_id:** has a data type of NUMBER.
- **role_id:** has a data type of NUMBER.

All attributes have a NOT NULL

The two lines under the check constraint defines a Composite Primary Key containing (**employee_id** + **department_id**)

The last few lines indicate a FOREIGN KEY constraint on **employee_id, department_id** and **role_id.** This means that

- The value in the employee_id column of EMPLOYEE_TABLEmust also exist as a primary key in the EMPLOYEE table.
- The value in the department_id column of EMPLOYEE_TABLEmust also exist as a primary key in the DEPARTMENT table.
- The value in the role_id column of EMPLOYEE_TABLEmust also exist as a primary key in the ROLE_TYPE table.

```
CREATE TABLE employee_role (
    employee_id     NUMBER NOT NULL, --NOT NULL constraint
    department_id   NUMBER NOT NULL, --NOT NULL constraint
    role_id         NUMBER NOT NULL,
    PRIMARY KEY ( employee_id,
                  department_id ),
    FOREIGN KEY ( employee_id )
        REFERENCES employee ( employee_id ),
    FOREIGN KEY ( department_id )
        REFERENCES department ( department_id ),
    FOREIGN KEY ( role_id )
        REFERENCES role_type ( role_id )
);
```

KEYWORD MEANINGS:

The following keywords are used through table creation queries and will be explained in the following table:

Keyword	Meaning
NUMBER:	Data type that ensures that only numeric values are entered into the respective column.
VARCHAR2	Data type which stands for variable character, it allows data of various lengths to be stored.
TIMESTAMP	Data type that allows for storing the date and the time.
DATE	Data type that allows for storing a date
PRIMARY KEY:	Keyword that enforces uniqueness ensuring that no two rows have the same value as a primary key
FOREIGN KEY Constraint	The foreign key in a table must exist as a primary key in the referred to table
Composite Primary Key	Consist of multiple columns that uniquely identifies each row in the table
NOT NULL:	Constraint which ensures that a user cannot insert a null value into the respective column
UNIQUE	Constraint which ensures that the user does not insert a value which has already been used in another row.
CHECK:	Is a constraint that limits the kind of data inserted into a certain column
ON DELETE CASCADE	Specifies that if a row is deleted from on table the corresponding rows in another table must also be deleted

Sequences

In a database a sequence generates unique increasing numbers. In our database we used sequences for our primary keys when inserting data into tables. This ensures that each row in the table has a unique identifier.

Action_Type Sequence

```
CREATE SEQUENCE action_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Department Sequence

```
CREATE SEQUENCE department_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Role_Type Sequence

```
CREATE SEQUENCE role_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Bank_Details Sequence

```
CREATE SEQUENCE bank_detail_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Product Sequence

```
CREATE SEQUENCE product_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Customer Sequence

```
CREATE SEQUENCE customer_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Employee Sequence

```
CREATE SEQUENCE employee_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Logged_Time Sequence

```
CREATE SEQUENCE logged_time_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Sequences(cont)

Purchase_Order Sequence

```
CREATE SEQUENCE purchase_order_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Shipment_Sequence

```
CREATE SEQUENCE shipment_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Supplier Sequence

```
CREATE SEQUENCE supplier_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Supplier Transaction Sequence

```
CREATE SEQUENCE supplier_transaction_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

Payment Sequence

```
CREATE SEQUENCE payment_id_seq
START WITH 1
INCREMENT BY 1
MINVALUE 1
NOMAXVALUE
CACHE 20 NOORDER
NOCYCLE;
```

SEQUENCES EXPLAINED

All the above sequences are set up in the same way and mean the following

START WITH 1: This were the sequence should start in all the cases above we start at one

INCREMENT BY 1: Each new number generated will be one more then the previous number

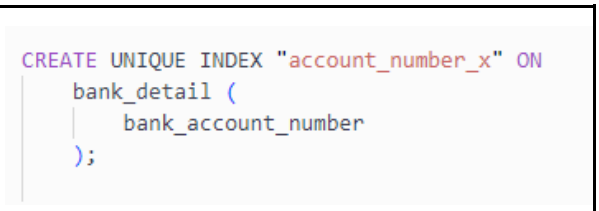
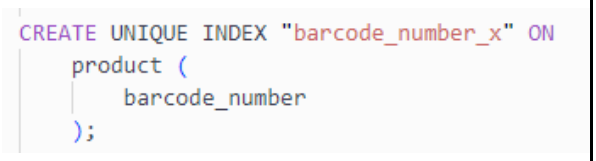
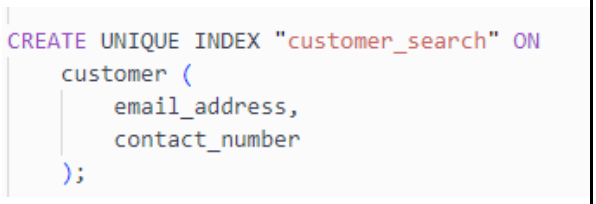
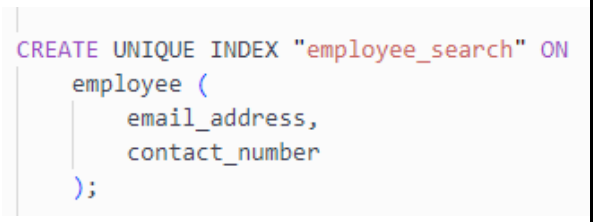
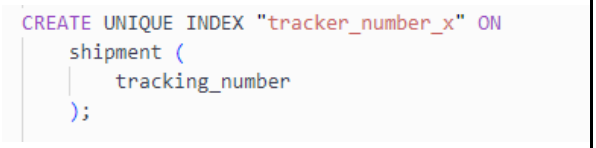
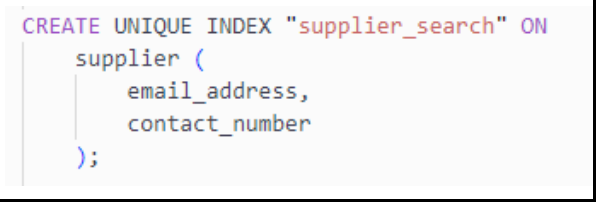
MINVALUE 1: the lowest value the sequence can generate is 1

NOMAXVALUE: this indicates that there is no upper limit however Oracle uses 10^28-1 as the max number.

CACHE 20: instructs the database to pre-allocate 20 numbers, which improves performance

NOORDER: the order in which numbers are generated may not be sequential

Indexes

Bank Account In this table we will be placing an index on bank_account_number to simplify the name for the user and to increase the speed to find the account number of a certain supplier or employee.This also helps to make sure the value is unique.	
Barcode Number In this index we will be placing a unique constraint on the barcode to make sure that two products cant have the same barcode.	
Customer Search In this index we create an index for customer email address and contact number, this will make sure that we can identify faster and spare the user time.	
Employee Search In this table we create an index for employee search using email_adress and contact number. This is to find a certain employee faster and more effective.	
Tracker number This index is there so that there cannot be two of the same tracking numbers in the system.	
Supplier Search The supplier search index is there to more easily identify the supplier using email address and contact number.	

Views

Order in progress view

The view named **orders_in_progress** gives us a filtered subset of data from the “PURCHASE_ORDER” table which is joined with the “ORDER_PRODUCT” table.

- **Description:** We start by retrieving certain columns from “PURCHASE_ORDER” and “ORDER_PRODUCT” that would be necessary to display for the user. The tables are then joined using an inner join linking each purchase order to its corresponding product. The rows are then filtered based on the order status received from “PURCHASE_ORDER”. If the status is cancelled, refunded, or completed they are not included as they are no longer active/in progress.
- **Use:** This view can be useful in monitoring all active orders. It can also be useful for inventory management as stakeholders can see which products are currently being ordered and in what quantities, providing a way to anticipate stock levels. Another good use is for customer service as it can aid a customer representative by addressing inquiries related to active orders.

```
CREATE OR REPLACE VIEW orders_in_progress AS
    SELECT
        po.purchase_order_id,
        po.employee_id,
        po.customer_id,
        po.order_placed_datetime,
        po.order_status,
        po.delivery_method,
        po.shipping_address,
        po.billing_address,
        po.payment_status,
        po.purchase_order_total,
        op.product_id,
        op.product_quantity
    FROM
        purchase_order   po
        JOIN order_product    op ON po.purchase_order_id = op.purchase_order_id
    WHERE
        po.order_status NOT IN (
            'Cancelled',
            'Refunded',
            'Completed'
        );
```

Employee Activity View

The view named **employee_activity_view** gives us a filtered subset of data from the “EMPLOYEE”, “EMPLOYEE_ROLE”, “DEPARTMENT”, and “ROLE_TYPE” tables. This view provides us with information on employee activity within the company.

- **Description:** We start by retrieving certain columns, from the tables named above, that would be necessary to display for the user. Using an inner join the tables are connected, ensuring that only employees with assigned roles and their corresponding departments are included.Left joins are then used to include logged time and action performed by employees. The joins are primarily performed based on the foreign key relationships between tables. The view then returns a subset providing insight into employee activities, including which employee belongs to which department and their roles. It also provides employees actions performed with their timestamp.
- **Use:** Managers can easily monitor the activity of employees, leading to managers identifying productivity levels, areas of improvement, and high performing employees.

```
CREATE OR REPLACE VIEW employee_activity_view AS
    SELECT
        e.employee_id,
        e.first_name
        || ' '
        || e.last_name AS employee_name,
        d.department_name,
        r.role_name,
        a.action,
        lt.logged_datetime
    FROM
        employee e
        JOIN employee_role er ON e.employee_id = er.employee_id
        JOIN department d ON er.department_id = d.department_id
        JOIN role_type r ON er.role_id = r.role_id
        LEFT JOIN logged_time lt ON e.employee_id = lt.employee_id
        LEFT JOIN action_type a ON lt.action_id = a.action_id;
```

Employee Work Hours View

The view named **employee_work_hours** gives us a filtered subset of data from the “EMPLOYEE”, “LOGGED_TIME”, and “ACTION_TYPE” tables. This view calculates and presents employee work hours and compensation for the current month.

- **Common table expressions:**
  - **work_logs:** Retrieves employee work logs for the current month and filters them according to logged datetime.
  - **in_out-pairs:** Aggregates the clock-in and clock-out times for every employee on each day of the month.
  - **work_sessions:** calculates the total work time for an employee's work session.

- **Description:** We start by retrieving certain columns, from the tables named above, that would be necessary to display for the user. A left join is used to connect “EMPLOYEE” with work_sessions on the employee ID to ensure every employee is included in the view. The data is then grouped by employee ID, first name, last name,

and their base salary. The total worked hours, overtime hours, and rates were then calculated. The sum of the worked intervals were converted to hours called **total_worked_hours**. The difference between the total_worked_hours and standard working hours a month, were calculated called **overtime_hours**. The base salary was then divided by the standard working hours a month, called **hourly_rate**. Finally the overtime rate is calculated as 1.5 times the hourly_rate.

- **Use:** This view provides essential information for calculating employee salaries and overtime pay. It also provides insight into the productivity of employees and their work habits. This view can also aid in labour expenses and cost optimization.

```
CREATE OR REPLACE VIEW employee_work_hours AS
    WITH work_logs AS (
        SELECT
            e.employee_id,
            e.first_name,
            e.last_name,
            lt.logged_datetime,
            at.action
        FROM
            employeee
            LEFT JOIN logged_time   lt ON e.employee_id = lt.employee_id
            LEFT JOIN action_type   at ON lt.action_id = at.action_id
        WHERE
            lt.logged_datetime >= trunc(sysdate, 'MM')
            AND lt.logged_datetime < add_months(trunc(sysdate, 'MM'), 1)
    ), in_out_pairs AS (
        SELECT
            employee_id,
            first_name,
            last_name,
            trunc(logged_datetime) AS log_date,
            MIN(
                CASE
                    WHEN action = 'Clock In' THEN
                        logged_datetime
                END
            ) AS clock_in_time,
            MAX(
                CASE
                    WHEN action = 'Clock Out' THEN
                        logged_datetime
                END
            ) AS clock_out_time
```

```
        FROM
            work_logs
        GROUP BY
            employee_id,
            first_name,
            last_name,
            trunc(logged_datetime)
    ), work_sessions AS (
        SELECT
            employee_id,
            first_name,
            last_name,
            clock_in_time,
            clock_out_time,
            ( clock_out_time - clock_in_time ) AS worked_time_interval
        FROM
            in_out_pairs
        WHERE
            clock_in_time IS NOT NULL
            AND clock_out_time IS NOT NULL
    )
```

```
SELECT
    e.employee_id,
    e.first_name,
    e.last_name,
    coalesce(SUM(EXTRACT(HOUR FROM ws.worked_time_interval) + EXTRACT(MINUTE FROM ws.worked_time_interval) / 60 + EXTRACT(SECOND
    FROM ws.worked_time_interval) / 3600), 0) AS total_worked_hours,
    greatest(coalesce(SUM(EXTRACT(HOUR FROM ws.worked_time_interval) + EXTRACT(MINUTE FROM ws.worked_time_interval) / 60 + EXTRACT
    (SECOND FROM ws.worked_time_interval) / 3600), 0) - 240, 0) AS overtime_hours,
    e.base_salary / ( 30 * 8 ) AS hourly_rate,
    ( e.base_salary / ( 30 * 8 ) ) * 1.5 AS overtime_rate,
    e.base_salary
FROM
    employee        e
    LEFT JOIN work_sessions   ws ON e.employee_id = ws.employee_id
GROUP BY
    e.employee_id,
    e.first_name,
    e.last_name,
    e.base_salary;
```

Extra Functionality: Triggers

Update product details trigger

The trigger named **trg_update_product_details** was designed to automatically update specific fields in the “PRODUCT” table, based on records added in the “SUPPLIER_TRANSACTION” table.

- **Trigger event:** The trigger executes on each insert operation taking place on “SUPPLIER_TRANSACTION”.
- **Compound trigger:** A compound trigger was implemented to allow us to define multiple sections of code to execute at different points in the execution of the statement
- **After each row:** This section updates the stock quantity of the corresponding product by adding the quantity of the inserted row, each time a new row is inserted into “SUPPLIER_TRANSACTION”.
- **After statement:** Within this section we calculate the average purchase price for every product from the “SUPPLIER_TRANSACTION” table. We then update the retail price field for every product in the “PRODUCT” table.
- **Loop:** It then runs through a loop iterating through the results, calculating a new retail price, and updating the retail price field in “PRODUCTS”.

```
CREATE OR REPLACE TRIGGER trg_update_product_details FOR
    INSERT ON supplier_transaction
COMPOUND TRIGGER
    v_avg_purchase_price   NUMBER;
    v_new_retail_price     NUMBER;
    AFTER EACH ROW IS BEGIN
        UPDATE product
        SET
            stock_quantity = stock_quantity + :new.product_quantity
        WHERE
            product_id = :new.product_id;
    END AFTER EACH ROW;
    AFTER STATEMENT IS BEGIN
        FOR rec IN (
            SELECT
                product_id,
                AVG(purchase_price) AS avg_price
            FROM
                supplier_transaction
            GROUP BY
                product_id
        ) LOOP
            v_new_retail_price := rec.avg_price * 1.05;
            UPDATE product
            SET
                retail_price = v_new_retail_price
            WHERE
                product_id = rec.product_id;
        END LOOP;
    END AFTER STATEMENT;
END trg_update_product_details;
/
```

Decrease stock trigger

The trigger named **decrease_stock_trigger** was designed to automatically update the stock quantity in the “PRODUCT” table whenever a new record is inserted into the “ORDER_PRODUCT” table.

- **Trigger event:** The trigger executes on each insert operation taking place on “ORDER_PRODUCT”.
- **After insert:** This section indicates to the trigger that it should only execute after the insert operation was done successfully.
- **Body:** The trigger starts by updating the stock quantity in “PRODUCT”, by subtracting the newly inserted product quantity. The product ID ensures that only the relevant product is updated.

```
CREATE OR REPLACE TRIGGER decrease_stock_trigger AFTER
    INSERT ON order_product
    FOR EACH ROW
BEGIN
    UPDATE product
    SET
        stock_quantity = stock_quantity - :new.product_quantity
    WHERE
        product_id = :new.product_id;

END;
/
```

Refill stock trigger

The trigger **refill_stock_trigger** was designed to automatically refill the stock quantity of a product when an order was cancelled or refunded, and the “PURCHASE_ORDER” table was updated.

- **Trigger event:** The trigger executes on each update operation taking place on “PURCHASE_ORDER”, when the order status column is updated.
- **After update:** This section ensures that the statement only executes after the update operation on order status is completed successfully.
- **Declare:** We declared some local variables namely “v_product_id”, to store the ID of the affected product, and v_product_quantity”, to store the quantity of the affected product.
- **Body:** The trigger starts by checking the order status and comparing it to the old and new values. If the status has changed, it then checks if the new status is either cancelled or refunded. The trigger then iterates through a loop using a cursor. For each cancelled or refunded product, it retrieves the product ID and product quantity, which it then updates the stock quantity in “PRODUCT”.

```
CREATE OR REPLACE TRIGGER refill_stock_trigger AFTER
    UPDATE OF order_status ON purchase_order
    FOR EACH ROW
DECLARE
    v_product_id         NUMBER;
    v_product_quantity   NUMBER;
BEGIN
    IF :old.order_status <> :new.order_status THEN
        IF :new.order_status = 'Cancelled' OR :new.order_status = 'Refunded' THEN
            FOR order_rec IN (
                SELECT
                    product_id,
                    product_quantity
                FROM
                    order_product
                WHERE
                    purchase_order_id = :new.purchase_order_id
            ) LOOP
                v_product_id := order_rec.product_id;
                v_product_quantity := order_rec.product_quantity;
                UPDATE product
                SET
                    stock_quantity = stock_quantity + v_product_quantity
                WHERE
                    product_id = v_product_id;
            END LOOP;
        END IF;
    END IF;
END;
/
```

Population of the Database

As we created tables we needed data to fill those tables. The data used to fill those tables each had its own data type depending on the data stored in that attribute.

We used the data types in our system:

- VARCHAR2 : We used a lot of different attributes with VARCHAR2 to help us read the data.
- Number : for data like stock quantity
- Timestamp : this was for our logged times so that we can use it for calculations
- Date : To see relevant dates in the system

There is a lot of different ways to import the data you can use scripts you wrote in python and C# to import the data or you can hard code it in. We used the hard code method and another method where we use csv files we created in excel to import the data. The hard code parts we used were there to showcase that we imported data and the csv files imported most of our data. The csv files when put into oracle changes to sql statements that you can run. This made the process a lot easier for us since we needed a lot of data to get good accurate testing results for our system

TABLE	SQL INSERT QUERY
Action_Type `INSERT INTO action_type ( action_id, action ) VALUES ( action_id_seq.nextval, 'Clock in' );`	Department `INSERT INTO department VALUES ( department_id_seq.nextval, 'Sales' );`
Role_Type `INSERT INTO role_type VALUES ( role_id_seq.nextval, 'Employee' );`	Bank_Detail `INSERT INTO bank_detail VALUES ( bank_detail_id_seq.nextval, 'ABC Bank', 'Savings', '12367890', '1234' );`
Product `INSERT INTO product VALUES ( product_id_seq.nextval, 'Copy Paper', 'Standard letter-size copy paper, 500 sheets', 'CP1001', 500, 9.99 );`	Employee `INSERT INTO employee VALUES ( employee_id_seq.nextval, 1, 'Rikus Gerhardus', 'Swart', 'rikus.swart@theoffice.com', '0605045379', 'password123', 20000 );`

Customer

```
INSERT INTO customer VALUES (
    customer_id_seq.nextval,
    'ABC Company',
    'John',
    'Doe',
    'john.doe@abccompany.com',
    1234567890,
    '123 Main Street'
);
```

```
INSERT INTO employee_role VALUES (
    1,
    1,
    1
); -- Employee in Sales department
```

Employee_Role

Logged_Time

```
INSERT INTO logged_time VALUES (
    logged_time_id_seq.nextval,
    1,
    1,
    TO_DATE('2024-05-14 09:00:00', 'YYYY-MM-DD HH24:MI:SS')
);
```

Shipment

```
INSERT INTO shipment VALUES (
    shipment_id_seq.nextval,
    2,
    TO_DATE('2024-01-10', 'YYYY-MM-DD'),
    'TRK234567890',
    'Alice Johnson',
    75.00
);
```

Payment

```
INSERT INTO payment VALUES (
    payment_id_seq.nextval,
    2,
    'Salary',
    6.2,
    'End-of-year bonus'
);
```

Supplier

```
INSERT INTO supplier VALUES (
    supplier_id_seq.nextval,
    1,
    'ABC Suppliers',
    'abc.suppliers@example.com',
    '1234567890'
);
```

Supplier _Transaction

```
INSERT INTO supplier_transaction VALUES (
    supplier_transaction_id_seq.nextval,
    4,
    4,
    TO_DATE('2024-05-04', 'YYYY-MM-DD'),
    'Transaction description 4',
    30,
    29.99
);
```

Order_Product

```
INSERT INTO order_product VALUES (
    4,
    4,
    12
);
```

Purchase_Order

```
INSERT INTO purchase_order VALUES (
    purchase_order_id_seq.nextval,
    1,
    1,
    TO_DATE('2024-05-14 10:00:00', 'YYYY-MM-DD HH24:MI:SS'),
    'Pending',
    'Standard',
    '123 Main St, Anytown',
    '456 Elm St, Anytown',
    'Pending',
    100.00
);
```

Queries

Criteria	Statement
Queries specifically solving company problems.	All Statements
Limitations of columns	All Statements
Limitations on rows	2, 5, 6, 8, 9, 10, 11, 12, 14, 16, 17, 18, 19, 23, 24, 26, 27, 28, 30, 31
Sorting	1, 2, 3, 4, 5, 6, 7, 8, 13, 15, 16, 17, 19, 23, 27, 28, 30, 31
LIKE, AND and OR	5, 8, 10, 14, 16, 17, 18, 19, 23, 26
Variables and character functions	18, 19, 20, 21, 31
Round or Trunc	5, 9, 10, 22
Date Functions	5, 8, 10, 23, 26
Aggregate Functions	4, 12, 19, 23, 24, 25, 26, 27, 28
Group by and Having	19, 27
Group By	4, 6, 9, 19, 23, 25, 28, 29, 30, 31
Joins	4, 5, 6, 7, 9, 10, 13, 18, 19, 23, 25, 27, 28, 29, 31
Sub-queries	29, 30
Extra Functionality	30

Statement 1:

Part of the company's objective was to have a more efficient way of tracking stock in real time.

- **Explanation**: This is achieved by selecting the product_name, product description, the barcode_number and current stock_quantity from the product table and ordering them by product name.
- **Use Case**: This enables effortless access to the current stock levels.

```
SELECT product_name, product_description, barcode_number, stock_quantity
FROM product
ORDER BY product_name ;
```

Statement 2:

Dunder Mifflin is a successful company that sells products that are constantly in demand. For this reason, it is essential to know which stock levels are low.

- **Explanation**: With the query below, all the products, along with their details and current stock_quantity, with a stock_quantity less than 10 are returned. These returned products are ordered by their stock_quantity in ascending order with the products lowest in stock first followed by the increasing stock_quantities.
- **Use Case**: This information can then be used to determine if it is necessary to order new stock from the supplier.

```
SELECT product_name, product_description, barcode_number, stock_quantity
FROM product
WHERE stock_quantity < 10
ORDER BY stock_quantity ASC;
```

Statement 3:

- **Explanation**: This statement calculates the current inventory value by multiplying the quantity of a product with its current retail price.
- **Use Case**: This information can then be used by the financial department to predict possible expenses and supplier orders for the coming month based on the company's current situation. For example if the inventory value of current stock for pencils are R50 000's worth, and the average sales of pencils a month equates to R25 000, we know we have two months worth of stock left.

```
SELECT product_name, stock_quantity, retail_price,
(stock_quantity * retail_price) AS inventory_value
FROM product
ORDER BY inventory_value DESC;
```

Statement 4:

Suppliers are in constant competition with each other and it is necessary to maintain a healthy relationship with dedicated suppliers.

- **Explanation**: The following statement evaluates each supplier and counts the amount of transactions that have taken place between them and the company. An inner join is used to join the supplier table with the supplier_transaction table to count all the applicable suppliers. The total orders are returned in return with the name of each supplier in ascending order from least transactions to most.
- **Use Case**: This can identify the suppliers that the company is mostly dependent on and can build a personal relationship with to acquire possible better future prices.

```
SELECT
    s.supplier_name,
    COUNT(st.supplier_transaction_id) AS total_orders
FROM
    supplier s
JOIN
    supplier_transaction st ON s.supplier_id = st.supplier_id
GROUP BY
    s.supplier_id, s.supplier_name
ORDER BY
    total_orders ASC;
```

Statement 5:

The following statement enables the company to display all the transactions with suppliers of the current month.

- **Explanation**: To achieve this, the supplier transaction table (that acts as a bridge entity) is joined with the product and supplier tables respectively. Then all the transactions are selected between two specific dates. In this case the function TRUNC is used to reset the current date to the first day of the month and the second date is the current date. This then all the suppliers and their associated transactions, along with the associated product that was supplied.
- **Use Case**: This statement is used to help keep track of the expenses of the current month.

```
SELECT
    s.supplier_name, p.product_name, su.transaction_datetime,
    su.transaction_description, su.product_quantity, su.purchase_price
FROM
    supplier_transaction su
JOIN
    supplier s ON su.supplier_id = s.supplier_id
JOIN
    product p ON su.product_id = p.product_id
WHERE
    su.transaction_datetime BETWEEN TRUNC(SYSDATE, 'MM') AND (SYSDATE)
ORDER BY
    su.transaction_datetime ASC;
```

Statement 6:

With such a variety of products sold and various moving components influencing the true reflection of sales.

- **Explanation**: The following statement filters out all unnecessary information to display the true sales of the company. The bridge table, order product, is joined with product and purchase order to select all the orders that have been completed. The quantity of products sold as well as the value of those sales are then summed and displayed along with the product name. The resulting information is then displayed in descending order from the highest value of total sales of a product, to the lowest.
- **Use Case**: This highlights the most popular and in demand products, along with products that are underperforming.

```
SELECT
    p.product_name,
    SUM(op.product_quantity) AS total_quantity_sold,
    SUM(op.product_quantity * p.retail_price) AS total_sales_value
FROM
    order_product op
JOIN
    product p ON op.product_id = p.product_id
JOIN
    purchase_order po ON op.purchase_order_id = po.purchase_order_id
WHERE
    po.order_status = 'Completed'
GROUP BY
    p.product_id, p.product_name
ORDER BY
    total_sales_value DESC;
```

Statement 7:

- **Explanation**: In this query we are joining the data from 3 tables, namely purchase_order, employee and customer in order to get more information about the order's details such as the employee who processed the order and the customer who placed the order as well as other important details about the order such as the current status of the order, the payment status and the purchase orders total.
- **Use Case**: From this query we can have a deeper understanding of the current workings of our business to ensure efficient processing and delivery goods and services. We can also track employee performance and how quickly orders are being processed.

```
SELECT
    e.first_name AS employee_first_name,
    e.last_name AS employee_last_name,
    po.customer_id,
    c.first_name AS customer_first_name,
    c.last_name AS customer_last_name,
    po.order_placed_datetime,
    po.order_status,
    po.delivery_method,
    po.shipping_address,
    po.billing_address,
    po.payment_status,
    po.purchase_order_total
FROM
    purchase_order po
JOIN
    employee e ON po.employee_id = e.employee_id
JOIN
    customer c ON po.customer_id = c.customer_id
ORDER BY
    po.order_placed_datetime DESC;
```

Statement 8:

- **Explanation**: This query retrieves all of the completed orders that took place between May and June and then orders the purchase order total from largest to smallest to show the order with the highest value first.
- **Use Case**: We wanted to see what our businesses performance was during this time period. It will be useful to know which products are the most sought after in order to prepare inventory for the following year.

```
SELECT order_placed_datetime, order_status, delivery_method, shipping_address,billing_address, payment_status, purchase_order_total
FROM purchase_order
WHERE EXTRACT(MONTH FROM order_placed_datetime) BETWEEN 5 AND 6
AND payment_status = 'Completed'
ORDER BY purchase_order_total DESC;
```

Statement 9:

- **Explanation**: We are retrieving the department_name, first_name and last_name of employees who have performed the ‘Clock in’ action since the start of operations. By joining multiple tables and using the WHERE clause we can filter to only display the records where the action_type is ‘Clock in’.
- **Use Case**: The primary purpose of the query was to track employee attendance and to log their total work hours for further queries, that would calculate the employee’s overtime hours and add to their payroll at the end of the month.

```
SELECT d.department_name, e.first_name, e.last_name
FROM employee e
INNER JOIN employee_role er ON er.employee_id = e.employee_id
INNER JOIN department d ON d.department_id = er.department_id
INNER JOIN logged_time lt ON lt.employee_id = e.employee_id
INNER JOIN action_type at ON at.action_id = lt.action_id
WHERE at.action = 'Clock in';
```

Statement 10:

- **Explanation**: We are displaying a list of all of the purchase orders that are currently taking place in the current month and sorts them from newest to oldest. It joins data from the purchase_order, customer, and employee tables and displays other critical information about the order such as its status and total.
- **Use Case**: A data dashboard can be set up in order to keep track of which employee completed which order and how many they completed in order to monitor their performance for this month. By doing this we can see if orders are being processed in a timely manner and if not to improve business operations.

```
SELECT c.company_name AS customer_name, e.first_name || ' ' || e.last_name AS employee_name,
po.order_placed_datetime, po.order_status, po.purchase_order_total
FROM purchase_order po
JOIN customer c ON po.customer_id = c.customer_id
JOIN employee e ON po.employee_id = e.employee_id
WHERE  po.order_placed_datetime BETWEEN TRUNC(SYSDATE, 'MM') AND (SYSDATE)
ORDER BY order_placed_datetime ASC;
```

Statement 11:

- **Explanation**: A query such as the one below showcases all of the purchase orders and the details of those orders where the order status is that of the Completed status. It could be improved by adding a date function to zone in on specific quarter of the businesses operations.
- **Use Case**: A business can draw conclusions about the preferred delivery method of their customers. They could also map out the shipping addresses of the customers to plan the delivery routes ahead of time and be more fuel efficient.

```
SELECT
    order_placed_datetime, delivery_method, shipping_address,
    billing_address, payment_status, purchase_order_total
FROM
    Purchase_Order
WHERE
    order_status = 'Completed';
```

Statement 12:

- **Explanation**: As explained in the video, the query counts the number of shipments that have been fulfilled by making use of aggregate functions such as COUNT(). It filters out all the records that are Null or have not yet been received by the customers and displays those who are NOT Null.
- **Use Case**: It deals with the final phase of the order which is the shipping process. We can see the reliability of the delivery process of the company by comparing if the shipment was delivered to the correct shipping address and received by the person or company who ordered it. We can also see if the same delivery mistakes are being made and how to improve upon it.

```
SELECT COUNT(*) AS num_shipments_fulfilled
FROM Shipment
WHERE Received_By IS NOT Null;
```

Statement 13:

- **Explanation**: It retrieves the last_name, first_name and role of all the employee's and then sorts them alphabetically by last_name. It joins 3 tables in order to get more information from other entities.
- **Use Case:** This query is useful for keeping track of which employee fills which role and how the roles are distributed across the hierarchical structure of the organisation as mentioned in Phase 1. Furthermore the assignment of new roles and promotions will be easier to do when the newest information about the business structure is available.

```
SELECT e.last_name,e.first_name, r.role_name
FROM employee e
JOIN employee_role er ON e.employee_id = er.employee_id
JOIN role_type r ON er.role_id = r.role_id
ORDER BY e.last_name;
```

Statement 14:

- **Explanation**: This query retrieves the first and last names of employees whose first names start with the letter 'R'. It makes use of the LIKE operator and the '%" percentage symbol to do so.
- **Use Case:** Would be to quickly find all the employee's based on a certain pattern of letters in order to perform delete, update or insert operations on their data.

```
SELECT first_name, last_name
FROM employee
WHERE first_name LIKE 'R%';
```

Statement 15:

- **Explanation**: The following statement helps to achieve an overview of the company's structure by using an inner join to join the employee with the employee role, department and role type respectively. This information is then ordered to display all departments and their employees together.
- **Use Case**: This is a commonly overlooked beneficial query that gives effortless insight into the organisational structure of a company.

```
SELECT
    d.DEPARTMENT_NAME AS department,
    r.ROLE_NAME AS role_type,
    e.FIRST_NAME AS employee_name,
    e.LAST_NAME AS employee_surname
FROM
    employee e
JOIN employee_role er ON e.employee_id = er.employee_id
JOIN department d ON er.department_id = d.department_id
JOIN role_type r ON er.role_id = r.role_id
ORDER BY
    d.department_name,
    r.role_name;
```

Statement 16:

- **Explanation**: This query retrieves the details of all the shipments that took place between the 1st of January and the 30th of June 2024. We display the shipment date, its tracking number who it has been received by and the shipment cost from the SHIPMENT table.
- **Use Case:** The purpose of this query for our company was to review our shipment costs and performance over the 1st half of the year. We can analyse our shipment costs in order to manage the expenses of the company.

```
SELECT shipment_date, tracking_number, received_by, shipment_cost
FROM SHIPMENT
WHERE SHIPMENT_DATE BETWEEN DATE '2024-01-01' AND DATE '2024-06-30';
```

Statement 17:

- **Explanation**: The query receives an employee's details based on the data inserted by the administration. A pop up will appear asking the user to insert data based on the first name and the last name. It can be vague or very specific as indicated by the LIKE operator. It makes use of substitution variables and to allow for the pop up dialog box.
- **Use Case:** Allows us to locate a specific employee in order to make changes to their personal information such as their last_names if they got married or their personal or company email addresses, or contact numbers but more importantly to adjust their base_salaries if they have been promoted or assigned a new role in the companies organisational structure.

```
SELECT first_name, last_name, email_address, contact_number, base_salary
FROM employee
WHERE first_name LIKE '%&first_name%'
AND last_name LIKE '%&last_name%';
```

Statement 18:

It is essential to minimise human input to maintain the consistency and integrity of the data in the database.

- **Explanation**: After supplying a barcode_number to the query below, all the details about the product are returned, including the product_name, product_description, barcode_number, stock_quantity and the current retail_price of the product. This can be effectively implemented by using PoS (Point-of_Sales) technology like a barcode scanner to dynamically query valuable information while minimising human input. Alternatively products can also be searched using the name of the product to return all products that match the product_search term or include similar characters.
- **Use Case**: This is useful to check the details of a product without having to type out the full, exact name of the product.

```
ACCEPT product_search PROMPT 'Enter the product bar code or product name: '
SELECT product_name,product_description,barcode_number,stock_quantity,retail_price
FROM product
WHERE barcode_number LIKE '%&product_search%'
   OR product_name LIKE '%&product_search%';
```

Statement 19:

- **Explanation**: This query also makes use of a substitution variable '&min_quantity' indicated by the '&' Ampersand sign. It allows the user to enter the information they wish to retrieve by providing a pop up dialog box. When prompted they must insert a quantity as an integer in order to display all the products whose quantity sold are more than the amount entered by the user.
- **Use Case:** By applying this query the company can focus on the products that have the highest total quantity sold. It will help them to make informed decisions on various areas of the business such as inventory control or even create promotions for those products retrieved that are in high demand.

```
SELECT
    p.product_name,
    SUM(op.product_quantity) AS total_quantity_sold
FROM
    order_product op
JOIN
    product p ON op.product_id = p.product_id
GROUP BY
    p.product_name
HAVING
    SUM(op.product_quantity) > (
        SELECT TO_NUMBER('&min_quantity') FROM dual
    )
ORDER BY
    total_quantity_sold DESC;
```

Statement 20:

- **Explanation**: This query initialises the first letter of the first_name field and makes the rest lowercase. It hides a certain part of the email_address for the privacy of customers in our database.
- **Use Case:** The information is hidden in order to prevent employees from knowing too much about a customer or when reports are generated that we still know whose email address it is but not being able to contact them.

```
SELECT  INITCAP(first_name),
CONCAT(RPAD(SUBSTR(email_address,1,3),9,'*'),
SUBSTR(email_address,
LENGTH(email_address)-11,
LENGTH(email_address))) AS "Secured_Email"
FROM employee;
```

Statement 21:

- **Explanation**: The following query receives data from the table shipment and rounds up the shipment cost to a whole number without decimals. The result is displayed in a new column called Rounded_Shipment_Cost
- **Use Case:** This query is used to generate a report showing the tracking numbers and the corresponding rounded shipment costs for all shipment without having to deal with decimals

```
SELECT tracking_number, ROUND(SHIPMENT_COST) AS ROUNDED_SHIPMENT_COST
FROM SHIPMENT;
```

Statement 22:

Thus far we have evaded technical business terminology but for the next part it is essential to understand what turnover rate is to fully grasp the value of the following statement. Turnover rate is basically the measure of how quickly products are sold relative to their average inventory levels. In our case, data is stored in various tables and must be fetched piece by piece to complete the whole picture.

- **Explanation**: Firstly, by using the WITH keyword, we create a Common Table Expression (CTE) that acts as a temporary table that can be referenced to later in the statement. This CTE is called total sales and calculates the total quantity of each product sold in the past month. Note that only the completed orders are selected, as they are counted as definite sales. Furthermore, we only select that transaction that took place in the last 30 days (independent of the month).

  We can now use a SELECT query to access information of the newly created total sales table and by using an inner join we can retrieve the necessary products from the products table. Now that we have all the information, we can simply calculate the turnover rate by dividing the quantity sold by the current stock quantity, with a null check to ensure if the value is 0, a null value is rather used to prevent division by 0.
- **Use Case**: The appropriate product details and rounded turnover rate is then displayed and can be used to better manage inventory and align stock levels with projected demand.

```
WITH total_sales AS (
    SELECT
        op.product_id,
        SUM(op.product_quantity) AS total_quantity_sold
    FROM
        order_product op
    JOIN
        purchase_order po ON op.purchase_order_id = po.purchase_order_id
    WHERE
        po.order_status = 'Completed'
        AND po.order_placed_datetime >= ADD_MONTHS(SYSDATE, -1)
    GROUP BY
        op.product_id
)
SELECT
    ts.product_id,
    p.product_name,
    ts.total_quantity_sold,
    p.stock_quantity,
    ROUND((ts.total_quantity_sold / NULLIF(p.stock_quantity, 0)),3) AS turnover_rate
FROM
    total_sales ts
JOIN
    product p ON ts.product_id = p.product_id
ORDER BY
    turnover_rate DESC;
```

Statement 23:

Purchase orders in the company can be in one of four states. Initially upon placing an order the order is Pending. The order can then also be Cancelled or Refunded and finally Completed.

- **Explanation:** In this statement we count all of the orders where the order status is completed to display a single number under the num_completed_orders column.
- **Use Case:** This can be performed on a regular basis to help with revenue tracking and determine the financial health of the company. For example when the company has completed 50 orders in their first month of operation, and the new completed orders number only rose to 75 in the second month of operation, we can clearly see a decrease of sales and productivity. Filtering the Completed orders from the rest is essential to the integrity and accuracy of the data, to display the real completed sales.

```
SELECT COUNT(*) AS num_completed_orders
FROM Purchase_Order
WHERE ORDER_STATUS = 'Completed';
```

Statement 24:

- **Explanation:** The following query returns a list of employees and their contact details, it also shows the number of completed orders associated with each employee. The first and last name is concatenated and called Full Name when running the query. The query joins together the two tables called Employee and Purchase_Order.
- **Use Case:** This can be used to identify the best performing employees or even to contact an employee if it was requested by an employee.

```
SELECT
    e.FIRST_NAME || ' ' || e.LAST_NAME AS FULL_NAME,
    e.EMAIL_ADDRESS,
    e.CONTACT_NUMBER,
    COUNT(CASE WHEN po.ORDER_STATUS = 'Completed' THEN 1 END) AS completed_orders_count
FROM
    Employee e
JOIN
    Purchase_Order po ON e.EMPLOYEE_ID = po.EMPLOYEE_ID
GROUP BY
    e.FIRST_NAME, e.LAST_NAME, e.EMAIL_ADDRESS, e.CONTACT_NUMBER;
```

Statement 25:

- **Explanation:** The following query is used to calculate the total amount spent on purchases from suppliers for the current month. The sum is used to get the total by multiplying product quantity and purchase price. The information is received from the supplier transaction table. The where clause filters the data retrieved based on the current month by using SYSDATE
- **Use Case:** This query can be used for monitoring monthly spendings of the company as well as a cost report. This allows us also to set up a performance analysis by comparing the total spent on purchases from suppliers across various months

```
SELECT SUM(product_quantity * purchase_price) AS Total_Purchases
FROM Supplier_Transaction
WHERE EXTRACT(MONTH FROM transaction_datetime) = EXTRACT(MONTH FROM SYSDATE)
AND EXTRACT(YEAR FROM transaction_datetime) = EXTRACT(YEAR FROM SYSDATE);
```

Statement 26:

The following statement allows for the selection of the products that have a quantity of more than 10 sold.

- **Explanation**: The information is simply gathered by joining the order product table with the products table and the purchase order table. Only the purchase orders that are then completed are selected and the total quantity is evaluated to meet the criteria of having more than 10 sold.
- **Use Case**: This information can be used to strategically plan promotions and discounts on popular products. Another use is to monitor the stock levels of these products as they are in high demand.

```
SELECT
    p.product_name,
    SUM(op.product_quantity) AS total_quantity_sold
FROM
    order_product op
JOIN
    product p ON op.product_id = p.product_id
JOIN
    purchase_order po ON op.purchase_order_id = po.purchase_order_id
WHERE
    po.order_status = 'Completed'
GROUP BY
    op.product_id, p.product_name
HAVING
    SUM(op.product_quantity) > 10
ORDER BY
    total_quantity_sold DESC;
```

Statement 27:

It is of utmost importance to establish a healthy customer relationship. Especially with regular customers that are a big part of the revenue flow of the company.

- **Explanation:** We select the first and last names of each customer, as well as all the purchase order totals along with their shipment costs (to reflect a true grand total). The information of the customer is easily selected from the customers table and by having access to the customer id, we can join the customer table with the purchase table to access the total amount. Having access to the purchase order id now, we can finally join the shipment table to access the last part that makes up the total cost.
- **Use Case:** This result of this statement is a list of grand totals displayed alongside the customers that placed that order. Expensive orders can then be identified where the company can then offer a personalised experience and enhance their relationship with the customer. This can also be used to calculate the average a customer spends per order for further financial analysis.

```
SELECT c.first_name,
       c.last_name,
       (po.purchase_order_total + s.shipment_cost) AS Total_Cost
FROM customer c
JOIN purchase_order po ON c.customer_id = po.customer_id
JOIN shipment s ON po.purchase_order_id = s.purchase_order_id;
```

Statement 28:

The paper company keeps track of various types of customers. Some customers are individuals looking to buy small quantities of stationery for personal use, but customers can also be companies that often order a large amount of stock.

- **Explanation**: Starting off by selecting the company name from the customer table, we must now access data that is located in the purchase order table. By making use of a sub-query, we select the join the customer table with the purchase order table on the customer id. For a customer id to be selected, its company field must not be null, indicating that the customer is indeed a company.
- **Usa Case**: We order the total purchase amount in descending order to identify companies that contribute the largest amount to our revenue first. Company relationships can be strengthened. Understanding which companies require the most resources from the paper company, we can better manage stock to suit their monthly needs and ensure their requirements can be met by possibly reserving stock for upcoming orders.

It is important to note that we are not incorporating shipment cost in the calculation of the total cost for a company, as this paper company does not deliver such large quantities of stock. Companies are required to collect their own orders.

```
SELECT
        company_name,
        total_purchase_amount
FROM (
    SELECT c.customer_id,
            c.company_name,
            SUM(po.purchase_order_total) AS total_purchase_amount
    FROM customer c
    JOIN purchase_order po ON c.customer_id = po.customer_id
    WHERE c.company_name IS NOT NULL
    GROUP BY c.customer_id, c.company_name, c.first_name, c.last_name
    ORDER BY total_purchase_amount DESC);
```

Statement 29:

- **Explanation**: This query allows us to display a prompt to the user for input asking them to enter a number in order to display the top or highest selling products in the business.

  It will then display the products_id, product_name, and total_quantity_sold of those top products from the order_product table. By performing a join we can know much more about the product attributes and details.
- **Use Case**: The company wants to analyse why these are the top products and to ensure that they are properly restocked for future business transactions. We can then plan marketing strategies based on these products to further increase the profitability of the company.

```
ACCEPT top_count PROMPT 'Enter the number of top products to display: '

SELECT *
FROM (
    SELECT
        op.product_id,
        p.product_name,
        SUM(op.product_quantity) AS total_quantity_sold
    FROM
        order_product op
    JOIN
        product p ON op.product_id = p.product_id
    JOIN
        purchase_order po ON op.purchase_order_id = po.purchase_order_id
    WHERE
        po.order_status = 'Completed'
    GROUP BY
        op.product_id, p.product_name
    ORDER BY
        total_quantity_sold DESC
)
WHERE
    ROWNUM <= &top_count;
```

Statement 30:

By making use of the employee work hours view we have previously created,not only are we able to quickly and easily access the frequently used data, but we are also able to separate some of the complexities of calculating the applicable values from this Select query.

- **Explanation**: In the following statement we are mainly selecting already calculated fields to display all the employees with a detailed breakdown of their current monthly performance.

  This performance breakdown tracks employees, overtime hours, hourly wage, overtime hourly wage and calculates the end of the month salary based on the overtime (performance) of the employee and also determines what percentage of the employees salary is made up of the bonus overtime. Their overtime hourly wage is based on the standard metric of 1.5 times their normal hourly wage.
- **Use Case**: This opens up a whole new world of insights by enabling Dunder Mifflin to ensure their workforce is as optimised as possible. This will also motivate employees to work harder or overtime to make up a greater bonus.

```sql
SELECT
    ew.first_name,
    ew.last_name,
    ROUND(ew.hourly_rate, 2) AS hourly_rate,
    ROUND(ew.overtime_rate, 2) AS overtime_rate,
    ew.total_worked_hours,
    ew.overtime_hours,
    ew.base_salary,
    ew.base_salary + ew.overtime_hours * ROUND(ew.overtime_rate, 2) AS total_pay,
    COALESCE(p.bonus_percentage, 0) AS bonus_percentage
FROM
    employee_work_hours ew
    LEFT JOIN payment p ON ew.employee_id = p.employee_id
ORDER BY
    ew.last_name,
    ew.first_name;
```

Statement 31:

Data is not worth anything if it is not processed to create valuable information. With the logged time data at our disposal, the following query breaks down an employee's break times logged over the past month. To avoid placing the burden on employees to constantly log the start and end of every task they perform, each break is given a set duration of minutes.

- **Explanation**: In the following query we create another CTE called break times. Using a case that acts like a simple conditional if in programming, we determine the action that is logged. Bathroom breaks are assigned 10 minutes, Smoke Breaks also 10 minutes and Lunch Breaks 60 minutes. These various breaks are then summed to make up the total of each section. Another field named total break hours then sums all the various categories of breaks as one and divides the sum by 60 to display the total of all the breaks in hours. This is done with the same principle case, used previously to identify the actions.

  Finally we acquire all of the remaining details from their associated tables to display a complete and robust breakdown of an employee's break log for the current month.
- **Use Case**: This information will be greatly beneficial to track employee habits and productivity and can even be used to calculate the actual hours an employee is working.

```sql
WITH break_times AS (
    SELECT
        e.employee_id,
        COALESCE(SUM( CASE WHEN at.action = 'Bathroom Break' THEN 10 ELSE 0 END ), 0) AS total_bathroom_break_minutes,
        COALESCE(SUM( CASE WHEN at.action = 'Smoke Break' THEN 10 ELSE 0 END ), 0) AS total_smoke_break_minutes,
        COALESCE(SUM( CASE WHEN at.action = 'Lunch Break' THEN 60 ELSE 0 END ), 0) AS total_lunch_break_minutes,
        ROUND(COALESCE(SUM(
            CASE
                WHEN at.action = 'Bathroom Break' THEN 10
                WHEN at.action = 'Smoke Break' THEN 10
                WHEN at.action = 'Lunch Break' THEN 60
                ELSE 0
            END
        ) / 60, 0), 2) AS total_break_hours
    FROM
        employee      e
        LEFT JOIN logged_time   lt ON lt.employee_id = e.employee_id
        LEFT JOIN action_type   at ON lt.action_id = at.action_id
    WHERE
        lt.logged_datetime >= TRUNC(SYSDATE, 'MM')
        AND lt.logged_datetime < ADD_MONTHS(TRUNC(SYSDATE, 'MM'), 1)
    GROUP BY
        e.employee_id
)
SELECT
    ew.first_name,
    ew.last_name,
    bt.total_bathroom_break_minutes,
    bt.total_smoke_break_minutes,
    bt.total_lunch_break_minutes,
    bt.total_break_hours
FROM
    employee_work_hours ew
    LEFT JOIN break_times bt ON ew.employee_id = bt.employee_id
ORDER BY
    ew.last_name,
    ew.first_name;
```

Statement 32:

- **Explanation**: The following statement, again, makes use of the created view to display all the employees' worked hours and overtime hours for the current month.
- **Use Case**: This is an added monitoring query that can be used to track employee performance. It is important for managers to get an oversight of their workforce so that employees can be rewarded for their dedication to their work.

```sql
SELECT
    ew.first_name,
    ew.last_name,
    ew.total_worked_hours,
    ew.overtime_hours
FROM
    employee_work_hours ew
ORDER BY
    ew.last_name,
    ew.first_name;
```

Statement 33:

To motivate employees in Dunder Mifflin, they must be willing to achieve better results from month to month. For this reason each month an employee's payment information must be updated to reflect the latest changes. This payment information includes very specific fields, namely bonus percentage and bonus description.

- **Explanation**: The bonus percentage is firstly calculated by dividing the total overtime bonus the employee has earned by the base salary of the employee and converting that to a percentage. With the view we have created, all the necessary fields are not recalculated but simply selected. Before updating the employees bonus description, we must first determine if the employee does in fact have a base salary assigned and if so, worked any overtime hours at all. This is done through the use of a case to fill in the appropriate description if the bonus cannot be calculated.

  Else if the employee does have a base salary, we can display the amount of overtime hours worked by the employee. When the employee did not work any overtime, we simply concatenate a 0 to the text "Overtime hours worked: ".
- **Use Case**: As previously mentioned, this performance evaluation metric allows employees to strive month to month to better their performance. Various employees may earn different salaries, and thus the bonus percentage allows us to standardise the playing field on employee performance, ultimately then we can identify certain outliers for example employee of the month, based on their performance.

```
UPDATE payment p
SET
    bonus_percentage = (
        SELECT
            CASE WHEN ew.base_salary = 0 THEN 0
                ELSE ROUND((ew.overtime_hours * ew.overtime_rate / ew.base_salary) * 100, 2)
            END
        FROM
            employee_work_hours ew
        WHERE
            ew.employee_id = p.employee_id
    ),
    payment_description = (
        SELECT
            CASE WHEN ew.base_salary = 0 THEN 'No bonus, base salary is 0'
                ELSE 'Overtime hours worked: ' || TO_CHAR(ew.overtime_hours)
            END
        FROM
            employee_work_hours ew
        WHERE
            ew.employee_id = p.employee_id
    )
WHERE
    EXISTS (
        SELECT
            1
        FROM
            employee_work_hours ew
        WHERE
            ew.employee_id = p.employee_id
    );
```