Phone-Pe

Data Analyst Interview Experience

CTC – ₹17 LPA

0-3 yoe

SQL Questions:

Let's assume a transactions table with the following relevant columns:

- transaction_id (VARCHAR/INT)
- user_id (VARCHAR/INT)
- amount (DECIMAL)
- transaction_date (DATETIME)
- status (VARCHAR) - 'SUCCESS' or 'FAILED'

- state (VARCHAR) - State where the transaction occurred
- city (VARCHAR) - City where the transaction occurred
- merchant_id (VARCHAR/INT)

1. Write a query to calculate the success rate of UPI transactions per state per day.

Concept: To calculate the success rate, we need to count the total transactions and the successful transactions for each state and date. The success rate is then (successful transactions / total transactions) * 100.

```
SELECT
    CAST(transaction_date AS DATE) AS transaction_day,
    state,
    COUNT(CASE WHEN status = 'SUCCESS' THEN 1 END) AS successful_transactions,
    COUNT(*) AS total_transactions,
    (CAST(COUNT(CASE WHEN status = 'SUCCESS' THEN 1 END) AS DECIMAL) * 100.0 / COUNT(*)) AS success_rate_percentage
FROM
    transactions
WHERE
    -- Assuming UPI transactions can be identified by a specific type or a flag, if not,
    -- this query assumes all transactions in the table are UPI transactions.
    -- If there's a 'payment_method' column: payment_method = 'UPI'
    status IN ('SUCCESS', 'FAILED') -- Only consider transactions that have a definite outcome
GROUP BY
    CAST(transaction_date AS DATE),
    state
ORDER BY
```

transaction_day,

state;

Explanation:

- CAST(transaction_date AS DATE): Extracts only the date part from the transaction_date timestamp to group transactions by day.
- COUNT(CASE WHEN status = 'SUCCESS' THEN 1 END): This is a conditional count. It counts only those rows where the status is 'SUCCESS'. If the condition is false, CASE returns NULL, and COUNT() ignores NULL values.
- COUNT(*): Counts all transactions for that specific transaction_day and state.
- (CAST(COUNT(CASE WHEN status = 'SUCCESS' THEN 1 END) AS DECIMAL) * 100.0 / COUNT(*)): Calculates the success rate. We CAST to DECIMAL to ensure floating-point division for accurate percentage calculation.
- GROUP BY CAST(transaction_date AS DATE), state: Groups the results by date and state.
- ORDER BY transaction_day, state: Orders the output for better readability.
- WHERE status IN ('SUCCESS', 'FAILED'): Ensures we only consider transactions that have a recorded outcome, excluding pending or other intermediary statuses.
- **Important Note:** The query assumes that the transactions table implicitly contains only UPI transactions, or that there's another column (e.g., payment_method) to filter for UPI. If there's a payment_method column, you'd add AND payment_method = 'UPI' to the WHERE clause.

2. Identify users who made transactions in three or more different cities within a month.

Concept: We need to group transactions by user and month, then count the distinct cities for each group. Finally, filter for groups where the distinct city count is 3 or more.

SELECT

user_id,

DATE_TRUNC('month', transaction_date) AS transaction_month -- For PostgreSQL/Redshift. Use DATE_FORMAT(transaction_date, '%Y-%m') for MySQL, or FORMAT(transaction_date, 'yyyy-MM') for SQL Server.

```
FROM
  transactions
GROUP BY
  user_id,
  DATE_TRUNC('month', transaction_date)
HAVING
  COUNT(DISTINCT city) >= 3;
```

Explanation:

- DATE_TRUNC('month', transaction_date): This function (common in PostgreSQL, Redshift) truncates the date to the beginning of the month, effectively grouping by month.
    - **Alternative for MySQL:** DATE_FORMAT(transaction_date, '%Y-%m')
    - **Alternative for SQL Server:** FORMAT(transaction_date, 'yyyy-MM') or DATEFROMPARTS(YEAR(transaction_date), MONTH(transaction_date), 1)
- GROUP BY user_id, DATE_TRUNC('month', transaction_date): Groups the data by each user for each distinct month.
- COUNT(DISTINCT city): Counts the number of unique cities a user transacted in within that specific month.
- HAVING COUNT(DISTINCT city) >= 3: Filters the grouped results to show only those user-month combinations where transactions occurred in 3 or more distinct cities.

3. From a transaction table, calculate the average transaction amount during peak hours (6 PM to 10 PM).

Concept: Filter transactions within the specified time range and then calculate the average of the amount column.

```
SELECT
  AVG(amount) AS average_peak_hour_transaction_amount
FROM
  transactions
```

```
WHERE
  CAST(transaction_date AS TIME) >= '18:00:00'
  AND CAST(transaction_date AS TIME) <= '22:00:00';
```

Explanation:

- CAST(transaction_date AS TIME): Extracts only the time part from the transaction_date timestamp.
- WHERE CAST(transaction_date AS TIME) >= '18:00:00' AND CAST(transaction_date AS TIME) <= '22:00:00': Filters the transactions to include only those that occurred between 6 PM (18:00:00) and 10 PM (22:00:00), inclusive.
- AVG(amount): Calculates the average of the amount for the filtered transactions.

4. Create a query to find merchants who had more than 5 failed transactions in a row.

Concept: This requires identifying consecutive failed transactions for each merchant. We can use window functions, specifically LAG or ROW_NUMBER combined with a common table expression (CTE), to track the status of previous transactions.

This is a more complex SQL problem, often solved using a technique called "gaps and islands".

```
WITH MerchantTransactions AS (
  SELECT
    merchant_id,
    transaction_date,
    status,
    ROW_NUMBER() OVER (PARTITION BY merchant_id ORDER BY transaction_date) AS rn,
    ROW_NUMBER() OVER (PARTITION BY merchant_id, status ORDER BY transaction_date) AS rn_status
  FROM
    transactions
),
```

```
FailedTransactionGroups AS (
  SELECT
    merchant_id,
    transaction_date,
    status,
    rn - rn_status AS group_id -- This creates a unique ID for consecutive sequences of the same status
  FROM
    MerchantTransactions
  WHERE
    status = 'FAILED'
)
SELECT
  DISTINCT merchant_id
FROM
  FailedTransactionGroups
GROUP BY
  merchant_id,
  group_id
HAVING
  COUNT(*) >= 6; -- We need 'more than 5', so >= 6 failed transactions in a row
```

Explanation:

1. **MerchantTransactions CTE:**
    - ROW_NUMBER() OVER (PARTITION BY merchant_id ORDER BY transaction_date) AS rn: Assigns a sequential number to each transaction for a given merchant based on its transaction_date.

- ROW_NUMBER() OVER (PARTITION BY merchant_id, status ORDER BY transaction_date) AS rn_status: Assigns a sequential number to each transaction for a given merchant *within the same status* based on transaction_date.

2. **FailedTransactionGroups CTE:**
    - rn - rn_status AS group_id: This is the core of the "gaps and islands" technique. When status is the same for consecutive rows, rn and rn_status will increment in sync, making their difference constant. This constant difference identifies a continuous "island" of identical statuses.
    - WHERE status = 'FAILED': We only care about failed transactions for this grouping.
3. **Final SELECT:**
    - GROUP BY merchant_id, group_id: Groups the failed transactions by merchant and by the identified consecutive failure group.
    - HAVING COUNT(*) >= 6: Filters these groups to find where the count of failed transactions in a consecutive sequence is 6 or more (i.e., "more than 5 in a row").
    - SELECT DISTINCT merchant_id: Returns the unique merchant_ids that satisfy the condition.

Python Questions:

1. Write a function to detect anomalies in transaction amounts using the IQR (Interquartile Range) method.

Concept: The IQR method defines outliers as data points that fall below Q1−1.5timesIQR or above Q3+1.5timesIQR, where Q1 is the first quartile (25th percentile), Q3 is the third quartile (75th percentile), and IQR=Q3−Q1.

Python

import pandas as pd

```
import numpy as np

def detect_anomalies_iqr(transaction_amounts):
    """
    Detects anomalies in a list or pandas Series of transaction amounts
    using the Interquartile Range (IQR) method.

    Args:
        transaction_amounts (list or pd.Series): A list or Series of numerical
                                                 transaction amounts.

    Returns:
        pd.Series: A boolean Series where True indicates an anomaly.
        tuple: A tuple containing (lower_bound, upper_bound) used for detection.
    """
    if not isinstance(transaction_amounts, (list, pd.Series)):
        raise TypeError("Input must be a list or pandas Series.")
    if len(transaction_amounts) < 4:
        # IQR method requires at least 4 data points for meaningful quartiles.
        # Although technically np.percentile can compute with fewer, it's less reliable.
        print("Warning: Too few data points for reliable IQR calculation. Returning empty anomalies.")
        return pd.Series([False] * len(transaction_amounts)), (None, None)

    # Convert to pandas Series for easier percentile calculation, if not already
    if not isinstance(transaction_amounts, pd.Series):
```

```
    amounts_series = pd.Series(transaction_amounts)
  else:
    amounts_series = transaction_amounts

  Q1 = amounts_series.quantile(0.25)
  Q3 = amounts_series.quantile(0.75)
  IQR = Q3 - Q1

  lower_bound = Q1 - 1.5 * IQR
  upper_bound = Q3 + 1.5 * IQR

  anomalies = (amounts_series < lower_bound) | (amounts_series > upper_bound)

  return anomalies, (lower_bound, upper_bound)

# Example Usage:
if __name__ == "__main__":
  # Sample transaction data
  transactions_data = [100, 110, 105, 120, 95, 130, 1500, 10, 115, 1000, 102, 98]
  amounts_series = pd.Series(transactions_data)

  anomalies, bounds = detect_anomalies_iqr(amounts_series)

  print("Transaction Amounts:")
  print(amounts_series)
  print("\nAnomalies Detected (True if anomaly):")
```

```
    print(anomalies)
    print(f"\nLower Bound: {bounds[0]:.2f}")
    print(f"Upper Bound: {bounds[1]:.2f}")

    print("\nAnomalous Transaction Amounts:")
    print(amounts_series[anomalies])

    # Another example with no anomalies
    transactions_no_anomalies = [50, 55, 60, 48, 62, 53]
    anomalies_no_anomalies, bounds_no_anomalies =
detect_anomalies_iqr(transactions_no_anomalies)
    print("\n--- Example with no anomalies ---")
    print("Transaction Amounts:")
    print(pd.Series(transactions_no_anomalies))
    print("\nAnomalies Detected (True if anomaly):")
    print(anomalies_no_anomalies)
    print(f"\nLower Bound: {bounds_no_anomalies[0]:.2f}")
    print(f"Upper Bound: {bounds_no_anomalies[1]:.2f}")
    print("\nAnomalous Transaction Amounts:")
    print(pd.Series(transactions_no_anomalies)[anomalies_no_anomalies])
```

Explanation:

1. **Import Libraries:** pandas for Series/DataFrame operations and numpy for numerical computations (though pandas.Series.quantile makes numpy less strictly necessary for this specific implementation, it's good practice for numerical tasks).
2. **detect_anomalies_iqr Function:**
    - Takes transaction_amounts as input.

- **Input Validation:** Checks if the input is a list or pandas Series and handles cases with very few data points, as IQR is less meaningful then.
- **Convert to Series:** Converts the input list to a pandas Series if it's not already, leveraging pd.Series.quantile() for easy calculation of quartiles.
- **Calculate Q1, Q3, and IQR:** Uses amounts_series.quantile(0.25) and amounts_series.quantile(0.75) to find the first and third quartiles. IQR is then calculated.
- **Calculate Bounds:** Defines the lower_bound and upper_bound using the 1.5timesIQR rule.
- **Identify Anomalies:** Creates a boolean Series anomalies where True indicates that the corresponding transaction amount is either below the lower_bound or above the upper_bound.
- **Return Values:** Returns the boolean anomalies Series and the calculated (lower_bound, upper_bound) for context.

2. Given two dictionaries of transaction IDs and amounts, write code to merge them and identify mismatches.

Concept: We need to combine the information from both dictionaries. For transactions with the same ID present in both dictionaries, we compare their amounts to find mismatches. Transactions present in only one dictionary can also be identified.

Python

```
def merge_and_identify_mismatches(dict1, dict2):
    """
    Merges two dictionaries of transaction IDs and amounts,
    and identifies transactions with mismatched amounts or
    transactions present in only one dictionary.

    Args:
        dict1 (dict): First dictionary (e.g., from source A),
                      keys are transaction IDs, values are amounts.
```

```
    dict2 (dict): Second dictionary (e.g., from source B),
                  keys are transaction IDs, values are amounts.

Returns:
    dict: A dictionary containing:
        - 'merged_transactions': A dictionary with all transactions,
                                 preferring dict1's value if no mismatch,
                                 or dict2's if dict1 is missing.
                                 For mismatches, it stores (amount_dict1, amount_dict2).
        - 'mismatched_amounts': A dictionary of transaction IDs and their
                                amounts from both dictionaries where they don't match.
                                Format: {id: (amount_dict1, amount_dict2)}
        - 'only_in_dict1': A dictionary of transactions present only in dict1.
        - 'only_in_dict2': A dictionary of transactions present only in dict2.
    """
    merged_transactions = {}
    mismatched_amounts = {}
    only_in_dict1 = {}
    only_in_dict2 = {}

    # Get all unique transaction IDs
    all_transaction_ids = set(dict1.keys()) | set(dict2.keys())

    for tx_id in all_transaction_ids:
        amount1 = dict1.get(tx_id)
        amount2 = dict2.get(tx_id)
```

```
    if amount1 is not None and amount2 is not None:
        # Transaction exists in both
        if amount1 != amount2:
            mismatched_amounts[tx_id] = (amount1, amount2)
            # For merged, you might choose one, or store both for review
            merged_transactions[tx_id] = (amount1, amount2)
        else:
            merged_transactions[tx_id] = amount1 # Amounts match, take from either
    elif amount1 is not None:
        # Only in dict1
        only_in_dict1[tx_id] = amount1
        merged_transactions[tx_id] = amount1
    else:
        # Only in dict2
        only_in_dict2[tx_id] = amount2
        merged_transactions[tx_id] = amount2

  return {
    'merged_transactions': merged_transactions,
    'mismatched_amounts': mismatched_amounts,
    'only_in_dict1': only_in_dict1,
    'only_in_dict2': only_in_dict2
  }

# Example Usage:
```

```
if __name__ == "__main__":
    transactions_source_a = {
        'tx1001': 150.75,
        'tx1002': 200.00,
        'tx1003': 50.25,
        'tx1004': 300.00, # This one will be mismatched
        'tx1005': 100.00, # Only in source A
    }

    transactions_source_b = {
        'tx1001': 150.75,
        'tx1002': 200.00,
        'tx1003': 50.25,
        'tx1004': 300.50, # Mismatched amount
        'tx1006': 75.00,  # Only in source B
        'tx1007': 25.00,  # Only in source B
    }

    results = merge_and_identify_mismatches(transactions_source_a,
transactions_source_b)

    print("Merged Transactions:")
    print(results['merged_transactions'])

    print("\nMismatched Amounts:")
    print(results['mismatched_amounts'])
```

```
print("\nTransactions Only in Source A:")
print(results['only_in_dict1'])

print("\nTransactions Only in Source B:")
print(results['only_in_dict2'])
```

Explanation:

1. **Initialization:** Create empty dictionaries to store the merged_transactions, mismatched_amounts, only_in_dict1, and only_in_dict2.
2. **all_transaction_ids:** Uses set union (|) to get all unique transaction IDs present in either dictionary. This ensures we iterate through every possible transaction.
3. **Iteration and Comparison:**
    - For each tx_id, it retrieves the amounts from dict1 and dict2 using dict.get(). get() is safer than direct indexing as it returns None if the key is not found, preventing KeyError.
    - **If tx_id exists in both:**
        - Compares amount1 and amount2. If they are different, it records the tx_id and both amounts in mismatched_amounts. The merged_transactions dictionary for this tx_id will store a tuple of both amounts for further investigation.
        - If amounts match, it adds the tx_id and the (matching) amount to merged_transactions.
    - **If tx_id exists only in dict1:** It adds the tx_id and amount1 to only_in_dict1 and merged_transactions.
    - **If tx_id exists only in dict2:** It adds the tx_id and amount2 to only_in_dict2 and merged_transactions.
4. **Return:** Returns a dictionary containing all the categorized results.

3. Explain the use of lambda functions and how they might help in transforming payment logs.

Explanation of Lambda Functions:

A lambda function in Python is a small, anonymous function defined with the lambda keyword. It can take any number of arguments but can only have one expression. The result of this expression is what the function returns.

Syntax: lambda arguments: expression

Key characteristics:

- **Anonymous:** They don't have a name like regular functions defined with def.
- **Single Expression:** They are limited to a single expression, which implies they cannot contain complex statements like if/else (though conditional expressions can be used), for loops, or multiple lines of code.
- **Concise:** Useful for quick, inline functions where a full def statement would be verbose.
- **Often used with higher-order functions:** They are frequently used with functions that take other functions as arguments, like map(), filter(), sorted(), apply() in pandas, etc.

How Lambda Functions Help in Transforming Payment Logs:

Payment logs often contain raw, unstructured, or semi-structured data that needs cleaning, parsing, and reformatting for analysis. Lambda functions are excellent for applying quick, single-line transformations across various elements of these logs, especially when combined with functions like map, filter, or pandas apply.

Here are some specific ways lambda functions can help:

1. **Parsing and Extracting Data:** Payment logs might have concatenated strings or embedded data that needs to be extracted.
    - **Example:** Extracting transaction type from a log message like "TXN_ID:12345, TYPE:UPI, AMT:100".

```
logs = [
  "TXN_ID:12345, TYPE:UPI, AMT:100",
  "TXN_ID:67890, TYPE:WALLET, AMT:250",
  "TXN_ID:11223, TYPE:NEFT, AMT:5000",
]
```

```
# Extracting the transaction type
transaction_types = list(map(lambda log: log.split('TYPE:')[1].split(',')[0].strip(), logs))
print(transaction_types) # Output: ['UPI', 'WALLET', 'NEFT']
```

2. **Type Conversion and Cleaning:** Converting string representations of numbers to actual numbers, or cleaning up whitespace.
    - **Example:** Converting string amounts to floats and removing currency symbols.

```
transaction_amounts_str = ["Rs 150.75", "Rs 200", "Rs 50.25"]
cleaned_amounts = list(map(lambda amt_str: float(amt_str.replace("Rs ", "")), transaction_amounts_str))
print(cleaned_amounts) # Output: [150.75, 200.0, 50.25]
```

3. **Filtering Logs:** Selecting specific logs based on certain criteria.
    - **Example:** Filtering for failed transactions.

```
payments = [
  {'id': 'tx1', 'status': 'SUCCESS', 'amount': 100},
  {'id': 'tx2', 'status': 'FAILED', 'amount': 50},
  {'id': 'tx3', 'status': 'SUCCESS', 'amount': 200},
  {'id': 'tx4', 'status': 'PENDING', 'amount': 75},
  {'id': 'tx5', 'status': 'FAILED', 'amount': 120},
]
failed_payments = list(filter(lambda p: p['status'] == 'FAILED', payments))
print(failed_payments)
# Output: [{'id': 'tx2', 'status': 'FAILED', 'amount': 50}, {'id': 'tx5', 'status': 'FAILED', 'amount': 120}]
```

4. **Creating New Features (with Pandas apply):** If payment logs are loaded into a pandas DataFrame, apply with lambda is very powerful for creating new columns based on existing ones.
    - **Example:** Categorizing transactions as 'small', 'medium', or 'large'.

```
import pandas as pd

df = pd.DataFrame(payments)

df['amount_category'] = df['amount'].apply(lambda x: 'Small' if x < 100 else ('Medium' if x < 200 else 'Large'))

print(df)

# Output:

#    id   status  amount amount_category

# 0  tx1  SUCCESS    100       Medium

# 1  tx2   FAILED     50        Small

# 2  tx3  SUCCESS    200        Large

# 3  tx4  PENDING     75        Small

# 4  tx5   FAILED    120       Medium
```

5. **Sorting Data:** Custom sorting based on a specific key or computed value.
    - **Example:** Sorting payments by amount.

```
sorted_payments = sorted(payments, key=lambda p: p['amount'])

print(sorted_payments)

# Output: [{'id': 'tx2', 'status': 'FAILED', 'amount': 50}, {'id': 'tx4', 'status': 'PENDING', 'amount': 75}, ...]
```

In summary, lambda functions provide a concise and elegant way to perform ad-hoc, simple transformations on payment log data, making code cleaner and often more readable for specific, one-off operations, especially within functional programming constructs or pandas operations.

Guesstimate Questions:

Guesstimate questions assess your ability to break down a large problem, make reasonable assumptions, and arrive at a logical (even if not perfectly precise) estimate. The process and assumptions are more important than the exact final number.

1. Estimate the number of UPI transactions PhonePe processes in India per day.

Approach:

1. **Start with Total UPI Transactions in India:** This is a crucial number. Let's assume you're aware of the overall UPI transaction volume in India or can make an informed guess.
    - **Fact Check (as of early 2025 data):** NPCI data shows UPI crossed 18.6 billion transactions in May 2025.
    - Monthly average daily transactions: $18.6 \text{textbillion} / 31 \text{textdays} \approx 0.6 \text{textbillion} = 600 \text{textmillion}$ transactions per day. Let's use **600 million daily UPI transactions** as a baseline.
2. **Estimate PhonePe's Market Share in UPI:** PhonePe is a dominant player in the UPI ecosystem.
    - **Assumption:** PhonePe and Google Pay are the top two, often vying for the #1 spot. Let's assume PhonePe holds roughly **45-50%** of the UPI market share by volume. Let's take **48%.**
3. **Calculate PhonePe's Daily UPI Transactions:**
    - PhonePe Daily UPI Transactions = Total Daily UPI Transactions in India * PhonePe's Market Share
    - PhonePe Daily UPI Transactions = $600 \text{textmillion} \text{times} 0.48$
    - PhonePe Daily UPI Transactions = 288textmillion

Estimated Number: PhonePe processes approximately **280-300 million UPI transactions per day** in India.

Key Assumptions Made:

- Total daily UPI transactions in India (based on NPCI data).
- PhonePe's market share in UPI (a critical assumption that should be justified with industry knowledge or a broad estimate).

How to refine if more time/data:

- Cite specific NPCI reports for recent UPI transaction volumes.

- Mention news articles or industry reports that discuss market share of various UPI apps.

2. How many users in tier-2 cities might use PhonePe for electricity bill payments monthly?

Approach:

1. **Estimate PhonePe's Total Monthly Active Users (MAU) in India:**
    - **Fact Check (from PhonePe Pulse Report - 2021, though it's older, good for demonstrating thought process):** PhonePe stated 13.3 Cr (133 million) monthly active users as of July 2021.
    - **Update/Assumption (for 2025):** Given the growth of digital payments, let's assume PhonePe has grown significantly. A more recent estimate (if available) would be better. Let's guesstimate around **250-300 million monthly active users** currently. Let's use **280 million MAU**.
2. **Estimate Percentage of PhonePe Users in Tier-2 Cities and Beyond:**
    - **Fact Check (from PhonePe Pulse Report - 2021):** "nearly 80% of our transactions come from tier 2, tier 3, tier 4 cities and beyond." This suggests a high user base in non-metro areas.
    - **Assumption:** Let's assume **70-75%** of PhonePe's MAU are from Tier-2 cities and beyond. Let's use **72%**.
3. **Estimate Percentage of Users Paying Electricity Bills:**
    - This is a general utility payment. Not every user pays electricity bills monthly through PhonePe. Some might pay annually, others might use different apps/methods, or live in areas with direct payment.
    - **Assumption:** Let's assume that among active users in tier-2+ cities, a significant portion uses PhonePe for bill payments. Perhaps **20-25%** of this segment actively pays electricity bills monthly through PhonePe. Let's use **22%**. This considers that not everyone is a homeowner, or they might prefer other payment methods.
4. **Calculate Estimated Users:**
    - Users in Tier-2+ cities = Total MAU * Percentage in Tier-2+ cities
        - 280textmilliontimes0.72=201.6textmillion users

    - Users paying electricity bills monthly = Users in Tier-2+ cities * Percentage paying electricity bills
        - 201.6textmilliontimes0.22approx44.35textmillion

Estimated Number: Approximately **40-45 million PhonePe users in tier-2 cities and beyond** might use the app for electricity bill payments monthly.

Key Assumptions Made:

- PhonePe's current total Monthly Active Users (MAU).
- The proportion of PhonePe's user base residing in Tier-2 cities and beyond.
- The conversion rate of these users to monthly electricity bill payments via PhonePe.

How to refine if more time/data:

- Try to find more recent MAU numbers for PhonePe.
- Consider average household electricity bill payment frequency (monthly/bi-monthly).
- Factor in internet penetration rates in Tier-2 cities.
- Consider competition for bill payments from other apps/direct payment methods.