Question 1: Data Cleaning and Quality Check

Quality Assessment Methodology

The quality assessment process in `data_quality_checks.ipynb` examines several dimensions of data quality:

1. Basic Dataset Information

We first analyse the basic structure of each dataset to understand:

- The shape and size of each table
- Column data types and non-null counts
- Uniqueness of values in key columns

This foundational step helps establish our baseline understanding of the data before deeper investigation.

2. Duplicate Detection

Duplicates in recruitment data can significantly skew metrics by counting the same events multiple times. Our duplicate check examines each table for exact duplicate rows:

```
# Check for duplicates
duplicate_count = df.duplicated().sum()
print(f"Duplicates: {duplicates} ({duplicates/len(df)*100:.2f}%)")
```

This validation is vital because duplicate records could lead to inflated vacancy counts or conversion metrics, resulting in misleading KPIs on the dashboard.

3. Missing Value Analysis

Missing data can compromise the integrity of time-to-fill and conversion metrics. We analyse missing values with:

```
# Check for missing values
missing = df.isnull().sum()
missing_pct = df.isnull().sum() / len(df) * 100
missing_info = pd.DataFrame({
    'Missing Values': missing,
    'Percentage': missing_pct
})
```

This analysis helps identify columns with high missing value rates, which can impact the accuracy of calculated metrics.

Identifying missing values is critical as it indicates potential data collection issues or integration problems between recruitment systems. For example, missing CLOSE_DATE values might affect time-to-fill

calculation accuracy.

4. Date Validation

A key focus of our quality assessment is date field validation. We check for:

- Invalid date formats
- Illogical date sequences (e.g., close dates before open dates)
- Future dates that are implausible

This validation is essential because recruitment metrics heavily rely on accurate timeline data to calculate KPIs like time-to-fill and conversion rates.

5. Outlier Detection

We employ statistical methods to identify outliers in critical metrics:

```
# Outlier detection example for time-to-fill
q1 = closed_reqs['time_to_fill'].quantile(0.25)
q3 = closed_reqs['time_to_fill'].quantile(0.75)
iqr = q3 - q1
lower_bound = max(0, q1 - 1.5 * iqr)
upper_bound = q3 + 1.5 * iqr
```

Detecting outliers is crucial as extreme values, such as unusually long time-to-fill periods, could distort average metrics and lead to misinterpretation of recruitment performance.

6. Status Progression Validation

For candidate data, we verify that status progressions follow a logical sequence:

```
def check_status_progression(group):
    # Sort by status start date
    sorted_statuses = group.sort_values('HISTORICAL_STATUS_START_DATE')

    prev_stage = 0
    valid_progression = True

    for status in sorted_statuses['CANDIDATE_HISTORICAL_STATUS']:
        # Logic to validate proper status progression
```

This validation ensures that candidate pipelines accurately reflect real-world recruitment workflows, which is essential for reliable conversion metrics.

7. Referential Integrity Checks

We verify proper relationships between tables, such as:

- All requisition IDs in the candidate table exist in the requisitions table

- All department IDs reference valid departments
- All candidate statuses correspond to valid status definitions

These checks ensure data consistency across the relational structure, which is foundational for accurate cross-table analytics.

Table Specific Cleaning

Requisitions Table

Issue: Missing RECRUITER Values

- Example: 30.65% of records (1,540 rows) had NULL values in the RECRUITER field.
- Impact: This would limit the ability to filter dashboard data by recruiter, one of the required filtering dimensions in the brief.
- Solution:

```
# a. Fix RECRUITER field (30.6% missing)
print("\\n1. Handling missing values:")
missing_recruiter = cleaned_data['requisitions']
['RECRUITER'].isnull().sum()
print(f"   - Missing RECRUITER values: {missing_recruiter}
({missing_recruiter/len(cleaned_data['requisitions'])*100:.2f}%)")

# Use RECRUITER_ID to fill in missing RECRUITER values where possible
recruiter_map = cleaned_data['requisitions'][cleaned_data['requisitions']
['RECRUITER'].notna()].groupby('RECRUITER_ID')
['RECRUITER'].first().to_dict()
print(f"   - Found {len(recruiter_map)} unique RECRUITER_ID to RECRUITER
mappings")

# Fill missing values using the mapping
before_fill = cleaned_data['requisitions']['RECRUITER'].isnull().sum()
cleaned_data['requisitions']['RECRUITER'] =
cleaned_data['requisitions'].apply(
    lambda row: recruiter_map.get(row['RECRUITER_ID']) if
pd.isnull(row['RECRUITER']) and row['RECRUITER_ID'] in recruiter_map else
row['RECRUITER'],
    axis=1
)
after_fill = cleaned_data['requisitions']['RECRUITER'].isnull().sum()
print(f"   - Filled {before_fill - after_fill} missing RECRUITER values")
print(f"   - Remaining missing RECRUITER values: {after_fill}
({after_fill/len(cleaned_data['requisitions'])*100:.2f}%)")
```

In this cleaning step, we identified RECRUITER as a critical field for filtering the dashboard data. Rather than arbitrarily imputing values, we leveraged the existing RECRUITER_ID to RECRUITER mappings in the dataset. By extracting these mappings, we were able to fill in all 1,540 missing values while preserving data integrity and relationships.

Issue: Missing CLOSE_DATE Values

- Example: 5.9% of records (296 rows) had NULL values in the CLOSE_DATE field.
- Impact: This would affect time-to-fill calculations and could lead to inaccurate reporting of active vacancies.
- Solution:

```
# b. Handle missing CLOSE_DATE (5.9% missing) - these are likely still
open
open_reqs = cleaned_data['requisitions'][cleaned_data['requisitions']
['CLOSE_DATE'].isnull()]
print(f"   - Found {len(open_reqs)} requisitions with missing CLOSE_DATE
(likely still open)")

# Check if these align with STATUS_IN
if 'STATUS_IN' in cleaned_data['requisitions'].columns:
    still_open = open_reqs[open_reqs['STATUS_IN'] == 'Open']
    incorrectly_marked = open_reqs[open_reqs['STATUS_IN'] != 'Open']

    print(f"   - {len(still_open)} of these are correctly marked as 'Open'
in STATUS_IN")
    if len(incorrectly_marked) > 0:
        print(f"   - {len(incorrectly_marked)} have missing CLOSE_DATE but
aren't marked as 'Open'")

        cleaned_data['requisitions'].loc[incorrectly_marked.index,
'STATUS_IN'] = 'Open'
```

For this issue, we took a different approach. Since missing CLOSE_DATE values likely indicate still-open requisitions, we chose to preserve these NULL values rather than imputing them. We also cross-validated with the STATUS_IN field to ensure data consistency, updating any requisitions with missing CLOSE_DATE values to show "Open" status if they weren't already marked as such.

Issue: Unusual NUMBER_OF_OPENINGS Values

- Example: We found 25 requisitions with more than 10 openings, with some having as many as 30 openings.
- Impact: Extreme values could skew vacancy metrics and potentially indicate data entry errors.
- Solution:

```
# 2. Check for unusual NUMBER_OF_OPENINGS
high_openings = cleaned_data['requisitions'][cleaned_data['requisitions']
['NUMBER_OF_OPENINGS'] > 10]
print(f"\n2. Found {len(high_openings)} requisitions with more than 10
openings")
```

For this issue, we chose to flag the unusual values rather than modify them. This preserves the original data while allowing analysts to be aware of potential outliers when interpreting results. The decision not to

modify these values is based on the understanding that some hiring initiatives might genuinely involve large numbers of openings.

Issue: Time-to-Fill Outliers

- Example: 197 requisitions had time-to-fill values that were statistical outliers.
- Impact: These outliers could significantly skew average time-to-fill metrics on the dashboard.
- Solution:

```
# b. Calculate and flag time-to-fill outliers
closed_reqs = cleaned_data['requisitions'][cleaned_data['requisitions']
['CLOSE_DATE'].notna()].copy()
if not closed_reqs.empty:
    closed_reqs['time_to_fill'] = (closed_reqs['CLOSE_DATE'] -
closed_reqs['OPEN_DATE']).dt.days

    # Calculate outlier bounds
    q1 = closed_reqs['time_to_fill'].quantile(0.25)
    q3 = closed_reqs['time_to_fill'].quantile(0.75)
    iqr = q3 - q1
    lower_bound = max(0, q1 - 1.5 * iqr)
    upper_bound = q3 + 1.5 * iqr

    # Add time_to_fill and outlier flag columns
    cleaned_data['requisitions']['time_to_fill'] =
closed_reqs['time_to_fill']
    cleaned_data['requisitions']['TIME_TO_FILL_OUTLIER'] = False
    cleaned_data['requisitions'].loc[
        closed_reqs[
            (closed_reqs['time_to_fill'] < lower_bound) |
            (closed_reqs['time_to_fill'] > upper_bound)
        ].index,
        'TIME_TO_FILL_OUTLIER'
    ] = True
```

Here, we employed the Interquartile Range (IQR) method to identify statistical outliers in time-to-fill values. Rather than removing or modifying these values, we added a new column TIME_TO_FILL_OUTLIER to flag them. This approach preserves the original data while providing a mechanism to filter out outliers during analysis if needed, which is particularly important for accurate KPI calculations on the dashboard.

Candidate Table Cleaning

Issue: Date Format Inconsistencies

- Example: Some date columns were not in datetime format, complicating temporal analysis.
- Impact: Inconsistent date formats could lead to incorrect time calculations and sequence validations.
- Solution:

```
# 2. Fix date columns
date_cols = ['SUBMISSION_DATE', 'HISTORICAL_STATUS_START_DATE',
'HISTORICAL_STATUS_END_DATE', 'LAST_MODIFIED_DATE']
for col in date_cols:
    # Convert to datetime, set errors to coerce to handle invalid dates
    if cleaned_data['candidate'][col].dtype != 'datetime64[ns]':
        cleaned_data['candidate'][col] =
pd.to_datetime(cleaned_data['candidate'][col], errors='coerce')
        print(f"   - Converted {col} to datetime format")
```

In this step, we standardized all date columns to the datetime64[ns] format, which ensures consistent handling of date operations throughout our analysis. The 'errors='coerce'' parameter converts invalid dates to NaT (Not a Time) values, which prevents errors while flagging potential data issues.

Issue: Illogical Date Sequences

- Example: 5,723 records had status start dates occurring before submission dates.
- Impact: These illogical date sequences could lead to negative duration calculations and confuse the candidate timeline analysis.
- Solution:

```
# 3. Handle records with illogical date sequences
# Check submission_date to historical_status_start_date
illogical_dates = cleaned_data['candidate'][
    (cleaned_data['candidate']['SUBMISSION_DATE'].notna()) &
    (cleaned_data['candidate']['HISTORICAL_STATUS_START_DATE'].notna()) &
    (cleaned_data['candidate']['HISTORICAL_STATUS_START_DATE'] <
cleaned_data['candidate']['SUBMISSION_DATE'])
]
print(f"\n2. Found {len(illogical_dates)} records where status start date
is before submission date")
print("   - These will be flagged but kept in the dataset")
cleaned_data['candidate']['ILLOGICAL_DATE_FLAG'] = False
cleaned_data['candidate'].loc[illogical_dates.index,
'ILLOGICAL_DATE_FLAG'] = True
```

For this issue, we added a flag column rather than correcting the dates. This approach acknowledges that we don't have enough information to determine which date is incorrect, but we still want to mark these records for potential exclusion or special handling in downstream analyses.

Issue: Missing Candidate IDs

- Example: 17 records had missing CANDIDATE_ID values.
- Impact: This could break relationships between tables and cause candidates to be excluded from pipeline analyses.
- Solution:

```
# 3. Handle missing CANDIDATE_ID values
missing_ids = cleaned_data['candidate'][cleaned_data['candidate']
['CANDIDATE_ID'].isna()]
print(f"\n3. Found {len(missing_ids)} records with missing CANDIDATE_ID")
print("   - These will be flagged but kept in the dataset")
cleaned_data['candidate']['MISSING_ID_FLAG'] = cleaned_data['candidate']
['CANDIDATE_ID'].isna()
```

Similar to other issues, we chose to flag these records rather than remove them. This preserves potentially valuable information while allowing analysts to filter them out if necessary for relationship-dependent analyses.

Issue: Invalid Status Values

- Example: 73,245 records contained status values not listed in the candidate_status table.
- Impact: These invalid statuses could lead to incorrectly calculated conversion rates and pipeline metrics.
- Solution:

```
# 4. Handle invalid status values
if 'candidate_status' in data:
    valid_statuses = set(data['candidate_status']
['CANDIDATE_HISTORICAL_STATUS'])
    invalid_statuses = cleaned_data['candidate']
[~cleaned_data['candidate']
['CANDIDATE_HISTORICAL_STATUS'].isin(valid_statuses)]
    cleaned_data['candidate']['INVALID_STATUS_FLAG'] =
~cleaned_data['candidate']
['CANDIDATE_HISTORICAL_STATUS'].isin(valid_statuses)
    print(f"\n4. Found {len(invalid_statuses)} records with invalid status
values")
    print("   - These have been flagged with INVALID_STATUS_FLAG")
```

Here we cross-referenced the status values with the canonical list in the candidate_status table. By flagging invalid values, we enable analysts to either exclude these records or potentially map them to valid statuses in future iterations of the analysis.

Issue: Pipeline Timing Outliers

- Example: 33,354 records had unusually long intervals between status changes.
- Impact: These outliers could distort average pipeline duration metrics and lead to incorrect process optimization decisions.
- Solution:

```
# 5. Handle pipeline timing outliers
# Calculate time between submission and status start
timing_data = cleaned_data['candidate'][
```

```
    (cleaned_data['candidate']['SUBMISSION_DATE'].notna()) &
    (cleaned_data['candidate']['HISTORICAL_STATUS_START_DATE'].notna())
].copy()

if not timing_data.empty:
    # Calculate days between submission and status start
    timing_data['days_to_status'] = (
        timing_data['HISTORICAL_STATUS_START_DATE'] -
        timing_data['SUBMISSION_DATE']
    ).dt.days

    # Only keep positive values (where status start is after submission)
    timing_data = timing_data[timing_data['days_to_status'] >= 0]

    # Calculate outlier bounds
    q1 = timing_data['days_to_status'].quantile(0.25)
    q3 = timing_data['days_to_status'].quantile(0.75)
    iqr = q3 - q1
    upper_bound = q3 + 1.5 * iqr

    # Add timing and outlier flag to main dataframe
    cleaned_data['candidate']['days_to_status'] =
timing_data['days_to_status']
    cleaned_data['candidate']['TIMING_OUTLIER'] = False
    cleaned_data['candidate
```

Issue: Status Progression Anomalies

Analysis of candidate pipelines revealed logical inconsistencies in status progressions:

- Found 120 candidate pipelines (0.07% of total) with illogical status sequences
- Examples include candidates marked as "Rejected" and then moved to "Interview" stages
- Added INVALID_PROGRESSION_FLAG to mark these records for special handling in reporting

Issue: Missing Candidate IDs

- 17 records were found with missing candidate IDs. These were flagged with MISSING_ID_FLAG but maintained in the dataset to preserve record counts.

Question 2: Table Transformation

Department Table Transformation

To address Question 2, I needed to transform the Department table into a more usable format. The hierarchical structure (Region > Area > Store) needs to be flattened into a more queryable format:

```
WITH hierarchy AS (
    SELECT
        d_store."DEPARTMENT_ID",
        d_store."DEPARTMENT_NAME",
```

```
        d_area."DEPARTMENT_NAME" AS "AREA",
        d_region."DEPARTMENT_NAME" AS "REGION"
    FROM department d_store
    JOIN department d_area ON d_store."PARENT_DEPARTMENT_ID" =
d_area."DEPARTMENT_ID"
    JOIN department d_region ON d_area."PARENT_DEPARTMENT_ID" =
d_region."DEPARTMENT_ID"
    WHERE d_store."DEPARTMENT_NAME" LIKE '% - SD'
      AND d_area."DEPARTMENT_NAME" LIKE 'Area % - SD'
      AND d_region."DEPARTMENT_NAME" LIKE 'Region % - SD'
)
SELECT * FROM hierarchy
ORDER BY "DEPARTMENT_ID";
```

This SQL transformation effectively flattens the hierarchy while preserving the relationships between store, area, and region levels. The result is a more user-friendly table that allows for efficient filtering in the dashboard.