01 · Explore Data Quality

This optional enhancement notebook inspects the raw National Water Plan dataset before any cleaning happens. It mirrors the guidance in `docs/DEVELOPMENT_RECOMMENDATIONS.md` by focusing on the diagnostic steps that inform how `scripts/data-cleaner.py` should be configured.

Notebook goals

- Understand baseline quality metrics (shape, missingness, duplicates)
- Inspect geographic coordinates to confirm valid ranges
- Review spill event coverage to surface outliers or data gaps
- Explore categorical health indicators for text-heavy columns

In [1]:
```
from __future__ import annotations

from pathlib import Path
import importlib.util
import sys

import dask.dataframe as dd
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

plt.style.use("seaborn-v0_8")
sns.set_theme(context="notebook", style="ticks")
pd.set_option("display.max_columns", 40)
pd.set_option("display.precision", 2)

PROJECT_ROOT = Path.cwd().resolve().parents[1]
DATA_PATH = PROJECT_ROOT / "data" / "national_water_plan.csv"
SCRIPTS_DIR = PROJECT_ROOT / "scripts"


def load_module(module_name: str, file_path: Path):
    if module_name in sys.modules:
        return sys.modules[module_name]

    spec = importlib.util.spec_from_file_location(module_name, file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules[module_name] = module
    spec.loader.exec_module(module)
    return module


data_loader = load_module("project_data_loader", SCRIPTS_DIR / "data-loader.
DataLoader = data_loader.DataLoader
```

```
DataConfig = data_loader.DataConfig
ExplorationReport = data_loader.ExplorationReport

data_cleaner = load_module("project_data_cleaner", SCRIPTS_DIR / "data-clear
WaterDataCleaner = data_cleaner.WaterDataCleaner
DataCleanerConfig = data_cleaner.DataCleanerConfig
```

```
Pydantic enhancements module not available. Using basic features only.
Pydantic enhancements module not available. Using basic features only.
Pydantic enhancements module not available. Using basic features only.
```

1. Load raw dataset with the production loader

The `DataLoader` handles schema validation, dtype optimization, and produces the same `ExplorationReport` structure the cleaning pipeline expects. Keeping the exploratory view aligned with production logic makes it easier to cross-reference findings.

In [2]:
```
data_config = DataConfig(filepath=str(DATA_PATH))
loader = DataLoader(data_config)
raw_ddf, exploration_report = loader.load_and_explore_data()

raw_preview = raw_ddf.head(5)
raw_preview
```

```
Duplicate check failed: _sum() got an unexpected keyword argument 'skipna'
Duplicate check failed in statistics: _sum() got an unexpected keyword argum
ent 'skipna'
Duplicate check failed: _sum() got an unexpected keyword argument 'skipna'
Duplicate check skipped (not supported for this operation)
High missing data percentage: 8.10%
```

Out[2]:

	ID	Water company	Site name	Longitude	Latitude	Receiving Environment	River Basin District	Ma C
0	AnW0001	Anglian Water	ABTHORPE TERMINAL PUMPING STATION	-1.06	52.11	Inland	Anglian	Bec
1	AnW0002	Anglian Water	SCHOOL LANE SPS ABY	0.12	53.28	Inland	Anglian	
2	AnW0003	Anglian Water	AKELEY (EX STW) PS	-0.97	52.04	Inland	Anglian	Bec
3	AnW0004	Anglian Water	ALDBOROUGH WATER RECYCLING CENTRE	1.25	52.86	Inland	Anglian	
4	AnW0005	Anglian Water	ALDEBURGH STW	1.61	52.14	Coastal	Anglian	

Baseline metadata

Use the exploration report to understand the big-picture dimensions and high-level warnings before diving into column-level diagnostics.

In [3]:
```
metadata = exploration_report.metadata
print(
    f"Rows: {metadata.rows:,}\n"
    f"Columns: {metadata.columns}\n"
    f"Memory (MB): {metadata.memory_usage:.2f}\n"
    f"Missing (%): {metadata.missing_values_percent:.2f}\n"
    f"Duplicate rows: {metadata.duplicate_rows:,}"
)

pd.DataFrame(
    {
        "type": ["errors", "warnings"],
        "messages": [exploration_report.errors, exploration_report.warnings]
    }
)
```

```
Rows: 14,187
Columns: 38
Memory (MB): 5.87
Missing (%): 8.10
Duplicate rows: 0
```

Out[3]:

	type	messages
0	errors	[]
1	warnings	[Duplicate check skipped (not supported for th...

Create a manageable in-memory sample

Large Dask frames are great for scale, but plotting and quick profiling is smoother with a pandas sample. Adjust `SAMPLE_ROWS` if you want a deeper slice.

In [4]:
```
SAMPLE_ROWS = 10_000
sample_pdf = raw_ddf.head(SAMPLE_ROWS, compute=True)

print(f"Sample rows: {len(sample_pdf):,} (first partitions only)")
sample_pdf.describe(include="all").transpose().head(10)
```

```
Sample rows: 10,000 (first partitions only)
```

Out[4]:

	count	unique	top	freq	mean	std	min	25%	50%	75
ID	10000	10000	AnW0001	1	NaN	NaN	NaN	NaN	NaN	Na
Water company	10000	7	Severn Trent	2466	NaN	NaN	NaN	NaN	NaN	Na
Site name	9734	9017	SOUTHEND - BELTON GARDENS PS	27	NaN	NaN	NaN	NaN	NaN	Na
Longitude	10000.0	NaN	NaN	NaN	-1.64	1.41	-6.34	-2.38	-1.61	-0.
Latitude	10000.0	NaN	NaN	NaN	52.63	1.41	49.92	51.47	52.6	53
Receiving Environment	10000	3	Inland	8395	NaN	NaN	NaN	NaN	NaN	Na
River Basin District	10000	10	Humber	1894	NaN	NaN	NaN	NaN	NaN	Na
Management Catchment	10000	185	Tame Anker and Mease	524	NaN	NaN	NaN	NaN	NaN	Na
Local Authority	10000	270	County Durham	541	NaN	NaN	NaN	NaN	NaN	Na
Water Body	10000	1927	Not part of a river WB catchment	1232	NaN	NaN	NaN	NaN	NaN	Na

2. Missing values and duplicates

Quantify missingness for critical fields to prioritize cleaning rules. The `DataCleanerConfig` uses these same column lists as guardrails.

In [5]:
```
base_config = DataCleanerConfig()
key_columns = [col for col in sorted(set(base_config.required_columns + base

missing_counts = (
    raw_ddf[key_columns]
    .isnull()
    .sum()
    .compute()
    .astype(int)
)

missing_df = (
    pd.DataFrame({"missing": missing_counts})
    .assign(percent=lambda df: (df["missing"] / metadata.rows) * 100)
    .sort_values("percent", ascending=False)
)

duplicate_series = raw_ddf.map_partitions(
    lambda part: pd.Series({"duplicates": int(part.duplicated().sum())}),
    meta=pd.Series(dtype="int64", name="duplicates"),
)
duplicate_count = int(duplicate_series.sum().compute())

print(f"Duplicate rows detected: {duplicate_count:,}")
missing_df.head(10)
```

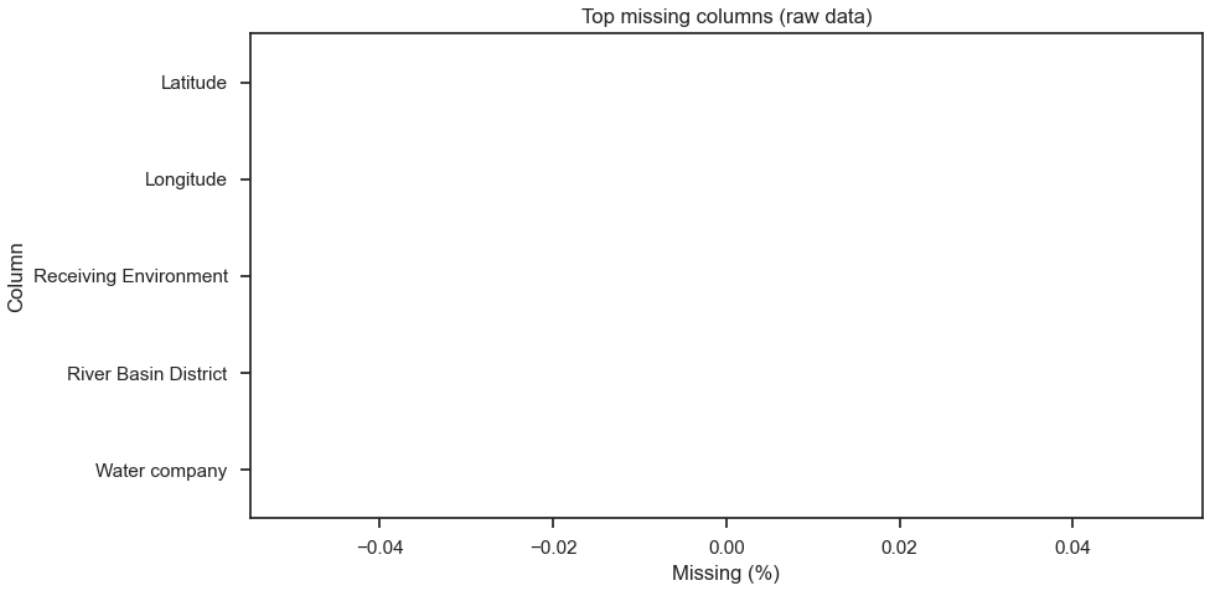
```
Duplicate rows detected: 0
```

Out[5]:

	missing	percent
Latitude	0	0.0
Longitude	0	0.0
Receiving Environment	0	0.0
River Basin District	0	0.0
Water company	0	0.0

In [6]:
```
fig, ax = plt.subplots(figsize=(10, 5))
(
    missing_df.head(12)[::-1]
    .plot(kind="barh", y="percent", legend=False, ax=ax, color="#3182bd")
)
ax.set_xlabel("Missing (%)")
ax.set_ylabel("Column")
ax.set_title("Top missing columns (raw data)")
```

```
plt.tight_layout()
plt.show()
```



3. Geographic coordinate diagnostics

Latitude/longitude validation removes invalid locations. Plotting a quick scatter exposes obvious geographic outliers before defining stricter bounds.

In [7]:
```
coord_cols = [col for col in ["Latitude", "Longitude"] if col in raw_ddf.col
coord_summary = (
    raw_ddf[coord_cols]
    .describe(percentiles=[0.25, 0.5, 0.75])
    .compute()
    .loc[["count", "min", "max", "mean", "std"]]
)

lat_range = (base_config.lat_min, base_config.lat_max)
lon_range = (base_config.lon_min, base_config.lon_max)

invalid_lat = (
    (~raw_ddf["Latitude"].between(lat_range[0], lat_range[1]))
    .sum()
    .compute()
    if "Latitude" in raw_ddf.columns
    else 0
)
invalid_lon = (
    (~raw_ddf["Longitude"].between(lon_range[0], lon_range[1]))
    .sum()
    .compute()
    if "Longitude" in raw_ddf.columns
    else 0
)

pd.DataFrame(
    {
```

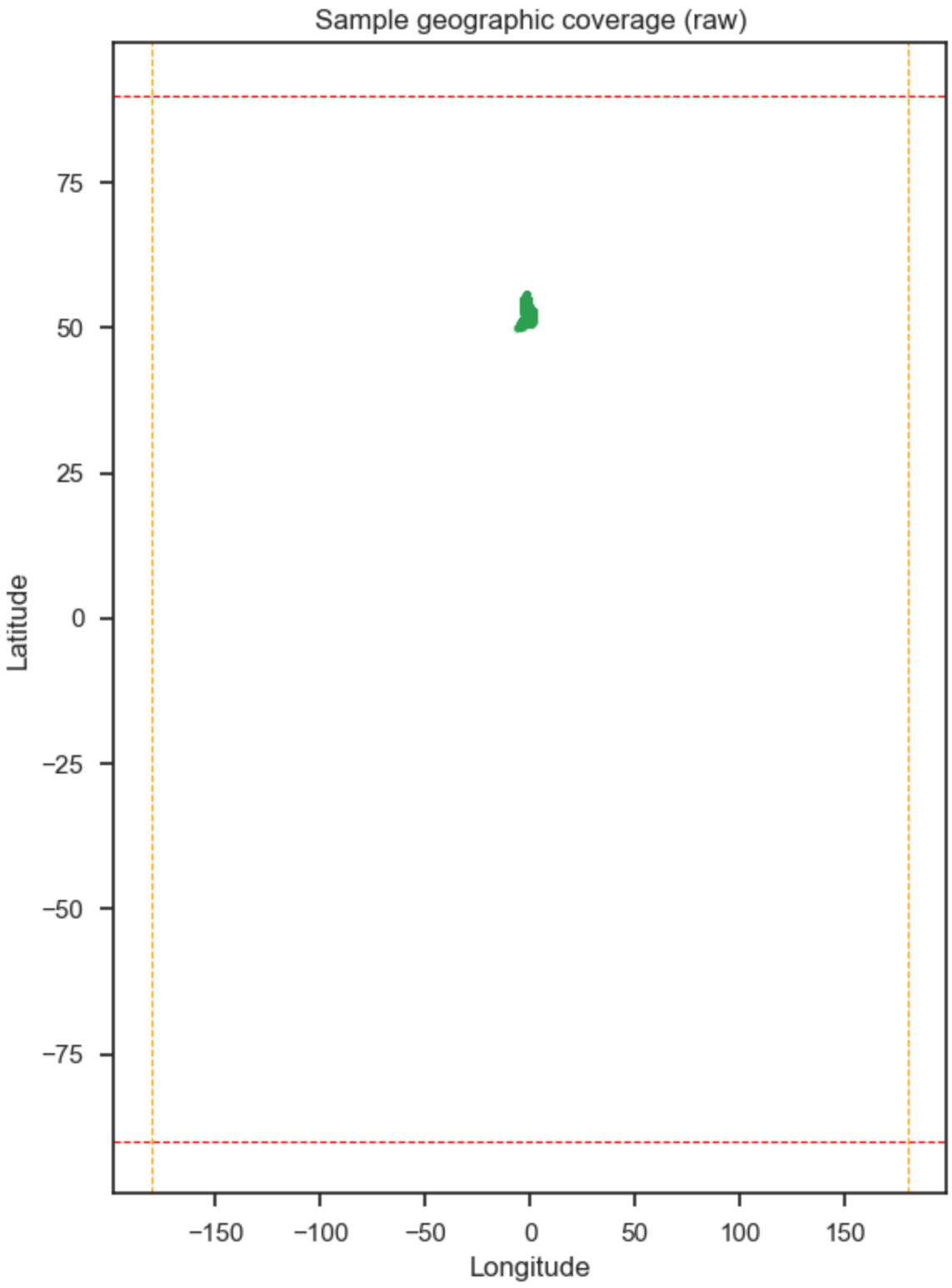
```
        "metric": ["latitude", "longitude"],
        "invalid_rows": [invalid_lat, invalid_lon],
        "min_config": [lat_range[0], lon_range[0]],
        "max_config": [lat_range[1], lon_range[1]],
    }
), coord_summary
```

Out[7]:

```
(      metric  invalid_rows  min_config  max_config
 0   latitude             0       -90.0        90.0
 1  longitude             0      -180.0       180.0,
        Latitude  Longitude
 count  14187.00   14187.00
 min       49.92      -6.34
 max       55.79       1.76
 mean      52.73      -1.73
 std        1.38       1.25)
```

In [8]:

```
geo_sample = sample_pdf.dropna(subset=coord_cols)
fig, ax = plt.subplots(figsize=(6, 8))
ax.scatter(
    geo_sample["Longitude"],
    geo_sample["Latitude"],
    s=10,
    alpha=0.4,
    linewidths=0,
    color="#31a354",
)
ax.set_title("Sample geographic coverage (raw)")
ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")
ax.axhline(lat_range[0], color="red", linestyle="--", linewidth=0.8)
ax.axhline(lat_range[1], color="red", linestyle="--", linewidth=0.8)
ax.axvline(lon_range[0], color="orange", linestyle="--", linewidth=0.8)
ax.axvline(lon_range[1], color="orange", linestyle="--", linewidth=0.8)
plt.tight_layout()
plt.show()
```



4. Spill event coverage

These columns often drive downstream analytics. Understanding missingness and value spread makes it easier to tune outlier handling and the `min_valid_spill_years` rule.

In [9]:
```
spill_cols = [col for col in base_config.spill_year_columns if col in raw_dd

spill_stats = (
    raw_ddf[spill_cols]
    .describe(percentiles=[0.5, 0.75, 0.9])
    .compute()
    .loc[["count", "mean", "std", "min", "50%", "75%", "90%", "max"]]
    .transpose()
    .rename(columns={"50%": "median"})
)

missing_spill = (
    raw_ddf[spill_cols]
    .isnull()
    .sum()
    .compute()
)

spill_stats = spill_stats.assign(
    missing=missing_spill,
    missing_pct=lambda df: (df["missing"] / metadata.rows) * 100,
)

spill_stats.head()
```

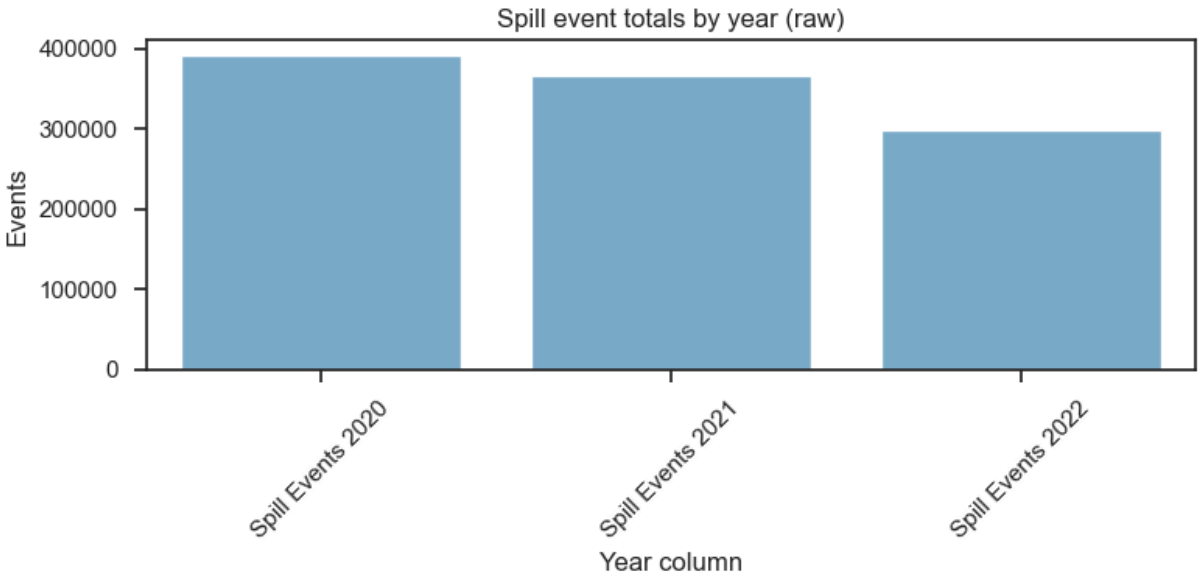
Out[9]:

	count	mean	std	min	median	75%	90%	max	missing	missing_pct
Spill Events 2020	11603.0	33.71	45.37	0.0	17.0	47.0	90.0	378.0	2584	18.21
Spill Events 2021	12399.0	29.53	37.64	0.0	16.0	43.0	77.0	361.0	1788	12.60
Spill Events 2022	13231.0	22.52	30.74	0.0	11.0	33.0	61.0	364.0	956	6.74

In [10]:
```
yearly_totals = (
    raw_ddf[spill_cols]
    .sum()
    .compute()
    .reset_index()
    .rename(columns={"index": "year", 0: "total_events"})
)

fig, ax = plt.subplots(figsize=(8, 4))
sns.barplot(
    data=yearly_totals,
    x="year",
    y="total_events",
    color="#6baed6",
    ax=ax,
)
```

```
ax.set_title("Spill event totals by year (raw)")
ax.set_xlabel("Year column")
ax.set_ylabel("Events")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



5. Text field health check

Whitespace trimming plus emptiness checks remove rows that lack contextual metadata. Review the dominant values to confirm expectations.

In [11]:
```
text_cols = [col for col in base_config.text_columns if col in sample_pdf.co
text_summary = []

for col in text_cols:
    series = sample_pdf[col].astype(str).str.strip()
    series = series.replace({"": pd.NA, "nan": pd.NA})
    missing = series.isna().sum()
    top_value = series.mode().iloc[0] if not series.mode().empty else None
    top_freq = int((series == top_value).sum()) if top_value else 0
    text_summary.append(
        {
            "column": col,
            "missing_pct": missing / len(series) * 100,
            "unique": series.nunique(dropna=True),
            "top_value": top_value,
            "top_freq": top_freq,
        }
    )

pd.DataFrame(text_summary).sort_values("missing_pct", ascending=False)
```

Out[11]:

	column	missing_pct	unique	top_value	top_freq
0	Water company	0.0	7	Severn Trent	2466
1	River Basin District	0.0	10	Humber	1894
2	Receiving Environment	0.0	3	Inland	8395

Takeaways

- Feed the missing value table into `DataCleanerConfig` thresholds (`missing_value_threshold`, `min_valid_spill_years`).
- Adjust coordinate ranges before running the pipeline if geographic outliers look legitimate.
- Use the spill event totals to calibrate outlier handling (`outlier_std_threshold`).
- Keep the text frequency table handy when reviewing rows that get dropped for empty metadata.