Plan2

思考時間: 55s >

結論：**一気通貫で組み込むべき**です。
CSV前提でも、**契約(Contract) → 取込(Ingest) → 検証(Quality Gates) → 正規化/特徴生成(Transform) → 因果用安全化(Causal-safe Prep) → 指標検査(Leakage/Overlap/PS分布)までをワンコマンド**化すれば、個人商用でも“壊れない前処理”になります。

3行サマリ（Takeaway）

1. **SSOT(単一真実源)のデータ契約**を先に置く。CSVはすぐ**Parquet**化（型・Null・カテゴリを固定）。
2. **DuckDB + dbt-duckdb**でSQL変換を標準化、**pandera/pytest**で品質をコード化、**Makefile**で一発実行。
3. 因果専用の**リーケージ防止・オーバーラップ検査**を前処理段に常設（失敗なら即Fail）。

ベストプラン（個人商用・CSV前提、最小で強い構成）

(1) Purpose / Non-goals

- **Purpose**：CSV取り込みから「因果推論に安全に渡せるデータ」を**再現可能・検査可能・拡張可能**に一気通貫化。
- **Non-goals**：
  - 本番DAG基盤（Airflow/K8s）は今回の最小構成に含めない（将来拡張点として分離）。
  - LLM前処理（テキスト大規模フィルタ）は別モジュール化。
  - 可視化UIの高機能化は対象外（CLI/レポート生成に限定）。

(2) 主要設計と代替案（理論とトレードオフ）

- **DuckDB + dbt-duckdb（採用）**
  - *理由*：ローカル単体で列指向・SQL最適化・CSV/Parquet自在。dbtで**スキーマ/テスト/変換**を宣言的に管理。
  - *Trade-off*：Sparkより分散スケールは狭いが、**個人商用〜中規模**では最も実効性が高い。
- **pandera + pytest（採用）**
  - *理由*：Python側で型・範囲・カテゴリの**Quality Gatesをコード化**。CIにも直結。
  - *代替*：Great Expectations（可視化豊富だが重量）、dbt testsのみ（SQL範囲の検査は得意だが、連続値分布/相関まで書くと煩雑）。
- **CSV→Parquet即時変換（採用）**
  - *理由*：CSVは**型揺れ/Null揺れ**が出やすい。早期Parquet化で**型・辞書エンコード**を固定。I/Oも高速化。
- **因果専用検査（採用）**
  - Propensity $\hat{e}(x)$ の**重なり(Overlap)**、共変量バランス(Love plot用指標)の**前処理段常設**。
  - *代替*：モデリング側に後出し → **ダメ**（データが壊れた後に気づく）。前処理で**Fail-fast**すべき。

(3) 境界（I/O・失敗時・拡張）

- **入力**：`data/raw/*.csv`（契約YAMLで列・型・必須/任意・カテゴリ定義）。
- **出力**：
  - `data/staged/*.parquet`（行一意キー付与・重複除去後）
  - `warehouse/warehouse.duckdb`（dbtのモデル格納）
  - `data/processed/causal_ready.parquet`
  - `artifacts/quality_report.md`, `artifacts/overlap_metrics.json`
- **失敗動作**：契約違反・品質閾値・オーバーラップ不足は**非ゼロ終了**（原因をstderrに明記）。
- **拡張点**：`profiles.yml`切替でSnowflake/BigQueryへ移行可。Airflow/Marquez(OpenLineage)は外付け可能。

具体構成（最小・監査可能・一発実行）

```
# 0) 依存と構成を作成（Fedora等のPOSIX前提）
mkdir -p ciq/{config,contracts,dbt_project,lib,scripts,tests} \
         ciq/{data/raw,data/staged,data/processed,artifacts,warehouse}

python -m venv .venv && source .venv/bin/activate
pip install -U pip
pip install duckdb dbt-duckdb pandas polars pyarrow pandera pydantic \
            scikit-learn pytest rich
```

```
# 1) 契約YAML（SSOT）
cat > ciq/contracts/dataset.yaml <<'YAML'
id_col: customer_id
time_col: event_time
treatment_col: treated
outcome_col: y
covariate_cols:
  - age
  - gender
  - rfm_score
  - prior_7d_spend
  - web_session_cnt
types:
  customer_id: int64
  event_time: datetime
  treated: int8     # 0/1
  y: float64        # 連続 or 0/1も可
  age: int16
  gender: category  # ["M","F","U"]
  rfm_score: float32
  prior_7d_spend: float32
  web_session_cnt: int16
constraints:
  not_null: [customer_id, event_time, treated]
  categories:
    gender: ["M","F","U"]
  ranges:
    age: [0, 110]
    rfm_score: [0.0, 1000.0]
    prior_7d_spend: [0.0, 1.0e6]
    web_session_cnt: [0, 10000]
leakage_forbidden_cols: [future_*, *_post]
YAML

# 2) DuckDB 用 dbt プロジェクト
cat > ciq/dbt_project/dbt_project.yml <<'YML'
name: ciq_dbt
version: 1.0
profile: ciq_duck
model-paths: ["models"]
models:
  ciq_dbt:
    +materialized: table
YML

mkdir -p ciq/dbt_project/models/{staging,curated,features}

cat > ciq/dbt_project/profiles.yml <<'YML'
ciq_duck:
  outputs:
    dev:
      type: duckdb
      path: ../warehouse/warehouse.duckdb
      threads: 4
  target: dev
YML

# 3) ステージング → 正規化 → 特徴（dbt SQL）
cat > ciq/dbt_project/models/staging/stg_events.sql <<'SQL'
select
  cast(customer_id as bigint) as customer_id,
  cast(event_time as timestamp) as event_time,
  cast(treated as integer) as treated,
  cast(y as double) as y,
  cast(age as smallint) as age,
  cast(gender as varchar) as gender,
  cast(rfm_score as real) as rfm_score,
  cast(prior_7d_spend as real) as prior_7d_spend,
```

```
  cast(web_session_cnt as integer) as web_session_cnt
from read_parquet('../data/staged/staged.parquet');
SQL

cat > ciq/dbt_project/models/curated/cur_events.sql <<'SQL'
with base as (
  select *,
         md5(concat(customer_id::varchar,'|',event_time::varchar)) as row_id
  from {{ ref('stg_events') }}
),
dedup as (
  select * from base qualify row_number() over(partition by row_id order by event_time desc)=1
),
clean as (
  select
    customer_id, event_time, treated, y,
    nullif(age, -1) as age,    -- sentinel to null
    upper(gender) as gender,
    rfm_score,
    coalesce(prior_7d_spend,0) as prior_7d_spend,
    web_session_cnt
  from dedup
)
select * from clean;
SQL

cat > ciq/dbt_project/models/features/causal_ready.sql <<'SQL'
-- 前処理の最終出力：因果安全（post列なし・欠損扱い統一・カテゴリ正規化）
select
  customer_id, event_time, treated, y,
  age,
  case when gender in ('M','F','U') then gender else 'U' end as gender,
  rfm_score, prior_7d_spend, web_session_cnt
from {{ ref('cur_events') }};
SQL

# 4) dbtテスト（NOT NULL, 一意, 受容値）
cat > ciq/dbt_project/models/schema.yml <<'YML'
version: 2
models:
  - name: cur_events
    tests:
      - not_null:
          column_name: customer_id
      - not_null:
          column_name: event_time
  - name: causal_ready
    tests:
      - not_null:
          column_name: treated
      - relationships:
          to: cur_events
          field: customer_id
      - accepted_values:
          column_name: gender
          values: ['M','F','U']
YML

# 5) Pythonライブラリ（契約検証・取込・因果安全化メトリクス）
cat > ciq/lib/contract.py <<'PY'
from __future__ import annotations
import yaml, re
from pathlib import Path
from typing import Dict, Any
import pandas as pd
import pandera as pa
from pandera import Column, DataFrameSchema, Check

def load_contract(path: str|Path) -> Dict[str, Any]:
```

```
    return yaml.safe_load(Path(path).read_text())

def build_schema(contract: Dict[str,Any]) -> DataFrameSchema:
    dtype_map = {
        "int8":"Int8","int16":"Int16","int32":"Int32","int64":"Int64",
        "float32":"Float32","float64":"Float64",
        "datetime":"datetime64[ns]","category":"category","bool":"bool"
    }
    columns = {}
    types = contract["types"]
    not_null = set(contract.get("constraints",{}).get("not_null",[]))
    ranges = contract.get("constraints",{}).get("ranges",{})
    cats = contract.get("constraints",{}).get("categories",{})
    for col, typ in types.items():
        checks = []
        if col in ranges:
            lo, hi = ranges[col]
            if lo is not None: checks.append(Check.ge(lo))
            if hi is not None: checks.append(Check.le(hi))
        if col in cats:
            checks.append(Check.isin(cats[col]))
        columns[col] = Column(dtype_map.get(typ, "object"),
                              nullable=(col not in not_null),
                              checks=checks)
    return DataFrameSchema(columns, coerce=True)

def forbid_leakage(df: pd.DataFrame, patterns:list[str]) -> None:
    for pat in patterns:
        rx = re.compile(pat.replace("*",".*"))
        bad = [c for c in df.columns if rx.fullmatch(c)]
        if bad:
            raise ValueError(f"Leakage columns detected: {bad}")
PY

cat > ciq/scripts/ingest.py <<'PY'
import sys, duckdb, pandas as pd, pyarrow as pa, pyarrow.parquet as pq
from pathlib import Path
from lib.contract import load_contract, build_schema, forbid_leakage

CONTRACT="ciq/contracts/dataset.yaml"
RAW_DIR=Path("ciq/data/raw")
STAGED=Path("ciq/data/staged/staged.parquet")

def main():
    contract = load_contract(CONTRACT)
    schema = build_schema(contract)
    files = sorted(RAW_DIR.glob("*.csv"))
    if not files:
        print("no CSV in ciq/data/raw", file=sys.stderr); sys.exit(2)

    dfs=[]
    for f in files:
        df = pd.read_csv(f)
        df = schema.validate(df, lazy=True)
        dfs.append(df)
    df_all = pd.concat(dfs, ignore_index=True)

    forbid_leakage(df_all, contract.get("leakage_forbidden_cols", []))

    # 型を固定化してParquetへ
    table = pa.Table.from_pandas(df_all, preserve_index=False)
    STAGED.parent.mkdir(parents=True, exist_ok=True)
    pq.write_table(table, STAGED)
    print(f"Wrote {STAGED} rows={len(df_all)}")

    # DuckDBに外部テーブルとして参照させる（dbtが読む）
    con = duckdb.connect("ciq/warehouse/warehouse.duckdb")
    con.execute("CREATE OR REPLACE VIEW staged_csv AS SELECT * FROM read_parquet(?)", [str(STAGED)])
```

```
if __name__=="__main__":
    main()
PY

cat > ciq/scripts/prepare_causal.py <<'PY'
import json, duckdb, numpy as np, pandas as pd
from pathlib import Path
from lib.contract import load_contract
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer

CONTRACT="ciq/contracts/dataset.yaml"
OUT_PARQUET=Path("ciq/data/processed/causal_ready.parquet")
METRICS_JSON=Path("ciq/artifacts/overlap_metrics.json")

def main():
    con = duckdb.connect("ciq/warehouse/warehouse.duckdb")
    df = con.execute("select * from causal_ready").fetch_df()

    c = load_contract(CONTRACT)
    X_cols = c["covariate_cols"]
    t_col = c["treatment_col"]

    # 欠損→単純代入→標準化（危険: 推定器に依存、ここは前処理として最小）
    imputer = SimpleImputer(strategy="median")
    scaler = StandardScaler(with_mean=True, with_std=True)

    X_num = df[X_cols].select_dtypes(include=[np.number]).values
    X_num = imputer.fit_transform(X_num)
    X_num = scaler.fit_transform(X_num)

    # 1) Propensityの粗推定（診断目的）
    lr = LogisticRegression(max_iter=1000)
    lr.fit(X_num, df[t_col].values)
    e_hat = lr.predict_proba(X_num)[:,1]
    df["ps_hat"] = e_hat

    # 2) オーバーラップ診断（0.05–0.95 に入る比率、極端値の質量）
    overlap = float(((e_hat>0.05)&(e_hat<0.95)).mean())
    tails = {"lt_0_01": float((e_hat<0.01).mean()), "gt_0_99": float((e_hat>0.99).mean())}

    # 3) 保存
    OUT_PARQUET.parent.mkdir(parents=True, exist_ok=True)
    df.to_parquet(OUT_PARQUET, index=False)
    METRICS_JSON.write_text(json.dumps({"overlap":overlap, "tails":tails}, indent=2))
    print("causal_ready saved,", "overlap=", overlap)

if __name__=="__main__":
    main()
PY

# 6) pytest（品質ゲートの最小検証）
cat > ciq/tests/test_contracts.py <<'PY'
from lib.contract import load_contract, build_schema
import pandas as pd
def test_schema_builds():
    c = load_contract("ciq/contracts/dataset.yaml")
    s = build_schema(c)
    df = pd.DataFrame({
        "customer_id":[1],
        "event_time":["2025-01-01 00:00:00"],
        "treated":[0],
        "y":[0.0],
        "age":[30],
        "gender":["M"],
        "rfm_score":[100.0],
        "prior_7d_spend":[0.0],
        "web_session_cnt":[0]
```

```
    })
    s.validate(df)  # 例外が出ないこと
PY

# 7) Makefile（ワンコマンド実行）
cat > Makefile <<'MK'
VENV=.venv
ACT=source $(VENV)/bin/activate

init:
    python -m venv $(VENV); $(ACT); pip install -U pip; \
    pip install duckdb dbt-duckdb pandas polars pyarrow pandera pydantic scikit-learn pytest rich

ingest:
    $(ACT); python ciq/scripts/ingest.py

dbt:
    $(ACT); cd ciq/dbt_project && dbt build --profiles-dir .

prepare:
    $(ACT); python ciq/scripts/prepare_causal.py

test:
    $(ACT); pytest -q

all: ingest dbt test prepare
MK
```

実行：

```
# 事前に raw CSV を投入
cp your.csv ciq/data/raw/events_2025_10.csv

# 一気通貫
make all
# 生成物:
# - ciq/data/staged/staged.parquet
# - ciq/warehouse/warehouse.duckdb（dbtモデル格納）
# - ciq/data/processed/causal_ready.parquet
# - ciq/artifacts/overlap_metrics.json
```

---

(4) 検証（pytest/metrics）

- `pytest`：契約→DataFrame変換→検証が通ること（**型/必須/範囲**）。
- `dbt build`：`stg → cur → causal_ready` が成功、`schema.yml`の**not_null/accepted_values**がPass。
- `overlap_metrics.json`：`overlap ≥ 0.6` 等の閾値を**商用閾値**としてCIで判定（低い場合はモデル適用不可 → 失敗）。
- （将来）Love plot 指標(標準化差分 |SMD|)を `artifacts/love_metrics.csv` に書き出し、**0.1以下**を目標。

---

(5) 数式（Wolfram系での要約）

- **重なり(Overlap)率**：$\mathrm{Overlap} = \mathbb{E}\left[1(0.05 < \hat{e}(X) < 0.95)\right]$
- **標準化差分(SMD)**：$\mathrm{SMD}_j = \dfrac{\bar{x}_{1j} - \bar{x}_{0j}}{s_j}$（前処理での共変量バランス；将来のメトリクス化）
- **失敗条件**：Overlapが低い or SMDが高い → **設計上、前処理でFail**（本プランのキモ）

What this does（この提案で“何がどう動くか”）

- **Goal**：CSVから**因果推論に安全**なデータを、**契約に基づく検査＆再現可能**な手順で**ワンコマンド生成**。
- **Inputs**：`ciq/data/raw/*.csv` と `contracts/dataset.yaml`（SSOT）。
- **Outputs/Side-effects**：Parquet固定化、DuckDB/DBTでの正規化テーブル、`causal_ready.parquet`、重なり診断JSON、dbtテスト結果。
- **Why this approach**：個人商用の現実解として**最小依存で最大の堅牢性**を確保（分散基盤不要・将来拡張容易）。
- **How to run/verify**：`make all` → pytest/dbt/診断メトリクスが**閾値未達なら非ゼロ終了**。CIにそのまま接続可。

よくある論点への先回り反論

- **Q: CSVのままでは？** → 型揺れ・Null揺れで地雷。**即Parquet化**が最安の保守。
- **Q: Sparkにしないの？** → 単独商用の初期は**DuckDB + dbt**が最速。将来BigQuery/Snowflake/Databricksへ**profiles**差替えで移行可。
- **Q: 品質検査はモデリング側で？** → **前処理でFail-fast**。壊れたデータをモデルに入れない。

将来拡張フック（必要になったら差すだけ）

- **Airflow**：`make`ターゲットを`airflow tasks`化。
- **OpenLineage/Marquez**：dbtのLineageを外部可視化。
- **Evidently**：データドリフトを `artifacts/drift_report.html` へ。
- **PII/DLP**：`ingest.py`にマスキング/トークン化フック。
- **Feature Store**：Feature Registry(YAML)→オンライン/オフライン一致。

Expert insight (Google/Meta/NASA級)

“因果の品質ゲートは統計学的に**可観測な失敗条件”として前処理で実装**せよ。
Overlapが低い、SMDが閾値超過、外れ値質量が大きい——これらは**データ段階の異常**であり、**学習段階のチューニングで救わない**。**Fail-fast**を前処理に内蔵することで、下流のA/B推定・政策最適化の**破綻を構造的に回避**できる。

必要なら上記スケルトンを**あなたの既存リポに最小差分で統合**するための`git diff`形式も即時出します。データ列仕様（`dataset.yaml`）だけ現場用に具体化すれば、そのままプロダクションに耐えます。

①PostgerSQLはどうですか。
②csvをアップする→分析なのですが、
前段階でどんなファイルの型でもアップ→変換→Paquetでアップロードの方が良いですか？

思考時間: 1m 4s

結論（3行サマリ）

1. **PostgreSQLは「制御面（メタデータ/ユーザ/監査/提供API）」に最適**。重い列指向分析はDuckDB/Parquet（将来はBQ/Snowflake/Databricks）で行う二層構成が個人商用の最短。
2. **アップロード形式は“絞ったホワイトリスト”→即Parquet正規化が最善**（CSV/TSV/JSONL/XLSX/Parquet＋gz）。「何でも可」はコストと事故率を爆増させる。
3. 既存スケルトンに**PostgreSQL連携（結果の配信先）＋多形式→Parquet変換**を最小差分で追加するのがベストプラン。

---

① PostgreSQLはどうか

- **得意**：ACID・高い安定性、ユーザ/認可、APIの背後DB、軽中量の集計、ジョブ・監査ログ、ML/因果の**結果配信先**（ダッシュボード/BIやRESTで配る）。
- **不得手**：巨大・列指向のETL/特徴生成/探索。Parquet/Icebergのような**列指向・圧縮・ファイル分割**が標準でない。拡張（FDW/columnar系）は手堅い運用には荷が重い。
- **推奨パターン（個人商用）**：
  - **データ面（Data Plane）**：Parquet（DuckDB/dbt）。
  - **制御面（Control Plane）**：PostgreSQL（認証、課金、実行履歴、レポート配信用の最終テーブル）。
  - つまり「**計算は列指向、配布はPostgres**」——無理にPostgresをDWH化しない。

② 「何でもアップ→変換→Parquet」か

- **はい（ただし“何でも”はやらない）**：プロダクトでは**入力形式をホワイトリストで限定**し、**即Parquet**で型・Null・カテゴリを固定化。
- **ホワイトリスト**：`{csv, tsv, jsonl, xlsx, parquet, feather} × {無圧/ .gz, .bz2}`。これで**99%の実務ニーズは足りる**。PDF/画像/任意バイナリは対象外（別モジュール）。
- **理由**：
  - CSV直解析は型揺れ・Null揺れの温床。

- Parquet固定で**速度↑/容量↓/型安定**、dbt/DuckDB/将来クラウドDWHへそのまま載せ替え可能。
- 入力上限・行数・文字コード・タイムゾーンを契約で固定し、**前処理でFail-fast**。

変更の目的 / Non-goals

- **目的**：既存スケルトンに**PostgreSQL配信用の出力導線**と**多形式→Parquetの前段**を追加し、一気通貫を強化。
- **Non-goals**：PostgresをDWH化しない。SparkやAirflowは将来拡張。PDF/画像/OCRなどは対象外。

主要設計と代替案（理論とトレードオフ）

- **採用：列指向（Parquet）をSSOT、DuckDB+dbtで変換**
  - *利点*：I/O効率、型固定、将来のIceberg/Delta移行が容易。
  - *トレードオフ*：単体運用（ローカル/単一ノード）スケールに限界→必要になればDWHへ昇格。
- **採用：PostgreSQLは配信用（結果・メタ・ユーザ）**
  - *利点*：API/ダッシュボードの標準DB、運用容易。
  - *トレードオフ*：重ETLは不得手。
- **代替**：全部Postgres（dbt-postgres）
  - *長所*：一元化。
  - *短所*：列指向でないため中長期でコスト増、運用チューニングが重い。**個人商用の速度と保守性に不利**。

境界（I/O・失敗時・拡張）

- **入力**：`ciq/data/raw/*.{csv,tsv,jsonl,xlsx,parquet,feather}{, .gz, .bz2}`（拡張子で判定＋魔法値で二段確認）。
- **出力**：`data/staged/*.parquet`（結合→`staged.parquet`）→dbt→`causal_ready.parquet`→（任意）**PostgresへCOPY**。
- **失敗**：契約違反（列欠落/型不一致/カテゴリ外）、サイズ超過、文字化け、TZ不一致、オーバーラップ低下→**非ゼロ終了**。
- **拡張**：S3/GCSにParquet保存、将来はIceberg/Delta、Airflow/Marquezを外付け。

変更差分（最小・監査可能）

依存追加（Postgres配布と多形式入出力）

```
source .venv/bin/activate
pip install psycopg2-binary openpyxl python-magic filetype # xlsx/型判定/PG連携用
```

1) 多形式→Parquet 変換スクリプトの追加

```
# ファイル作成
mkdir -p ciq/scripts ciq/sql
cat > ciq/scripts/convert_any_to_parquet.py <<'PY'
import os, sys, gzip, bz2, io, pandas as pd, pyarrow.parquet as pq, pyarrow as pa
from pathlib import Path
import magic
from lib.contract import load_contract, build_schema

RAW=Path("ciq/data/raw")
STAGED=Path("ciq/data/staged/staged.parquet")
CONTRACT="ciq/contracts/dataset.yaml"

def _open(path: Path):
    if str(path).endswith(".gz"):  return io.TextIOWrapper(gzip.open(path, "rb"))
    if str(path).endswith(".bz2"): return io.TextIOWrapper(bz2.open(path, "rb"))
    return open(path, "r", newline="", encoding="utf-8", errors="replace")

def load_one(path: Path) -> pd.DataFrame:
    mime = magic.from_file(str(path), mime=True)
    p = str(path).lower()
    if p.endswith((".csv",".csv.gz",".csv.bz2")):
        return pd.read_csv(_open(path))
    if p.endswith((".tsv",".tsv.gz",".tsv.bz2")):
        return pd.read_csv(_open(path), sep="\t")
    if p.endswith((".jsonl",".jsonl.gz",".jsonl.bz2",".ndjson")):
        return pd.read_json(_open(path), lines=True)
    if p.endswith(".xlsx"):
```

```
        return pd.read_excel(path)
    if p.endswith(".parquet"):
        return pd.read_parquet(path)
    if p.endswith(".feather"):
        return pd.read_feather(path)
    raise ValueError(f"unsupported file type: {path} (mime={mime})")

def main():
    files = sorted([p for p in RAW.iterdir() if p.is_file()])
    if not files:
        print("no input files", file=sys.stderr); sys.exit(2)
    c = load_contract(CONTRACT); schema = build_schema(c)
    dfs = []
    for f in files:
        df = load_one(f)
        df = schema.validate(df, lazy=True)  # 型・範囲・カテゴリを固定
        dfs.append(df)
    import pandas as pd
    df_all = pd.concat(dfs, ignore_index=True)
    table = pa.Table.from_pandas(df_all, preserve_index=False)
    STAGED.parent.mkdir(parents=True, exist_ok=True)
    pq.write_table(table, STAGED)
    print(f"Wrote {STAGED} rows={len(df_all)}")

if __name__ == "__main__":
    main()
PY
```

2) PostgreSQLへ結果を配布（COPY取り込み）

```
cat > ciq/sql/pg_schema.sql <<'SQL'
create table if not exists causal_ready (
  customer_id     bigint,
  event_time      timestamp with time zone,
  treated         integer,
  y               double precision,
  age             smallint,
  gender          text,
  rfm_score       real,
  prior_7d_spend  real,
  web_session_cnt integer,
  ps_hat          double precision
);
SQL

cat > ciq/scripts/export_for_pg.py <<'PY'
import duckdb, pandas as pd, gzip
from pathlib import Path
OUT=Path("ciq/artifacts/causal_ready.csv.gz")
con = duckdb.connect("ciq/warehouse/warehouse.duckdb")
df = con.execute("select * from causal_ready").fetch_df()
OUT.parent.mkdir(parents=True, exist_ok=True)
with gzip.open(OUT, "wt", encoding="utf-8") as f:
    df.to_csv(f, index=False)
print("exported", OUT)
PY
```

3) Makefileの追加ターゲット

```
# 追記
cat >> Makefile <<'MK'

convert:
    $(ACT); python ciq/scripts/convert_any_to_parquet.py

pg_create:
    psql "$$PG_DSN" -f ciq/sql/pg_schema.sql
```

```
pg_load:
    $(ACT); python ciq/scripts/export_for_pg.py
    zcat ciq/artifacts/causal_ready.csv.gz | psql "$$PG_DSN" -c "\copy causal_ready FROM STDIN WITH (FORMAT csv, HEADER true)"

# 多形式→Parquet→dbt→検証→因果準備→(任意)PG配布
all_plus_pg: convert dbt test prepare pg_create pg_load
MK
```

> *環境変数例：*
> `export PG_DSN="postgresql://user:pass@localhost:5432/ciqdb?sslmode=disable"`

実行コマンド（例）

```
# 1) 依存追加
source .venv/bin/activate
pip install psycopg2-binary openpyxl python-magic filetype

# 2) 入力（複数形式OK）
cp a.csv ciq/data/raw/
cp b.jsonl.gz ciq/data/raw/
cp c.xlsx ciq/data/raw/

# 3) 一気通貫（PG配布まで）
make all_plus_pg
```

検証（pytest/metrics/整合）

- **契約検証**：`convert` 実行時に `pandera` が型・範囲・カテゴリを検査（例外でFail）。
- **dbt tests**：`dbt build` の `not_null/accepted_values/relationships` がPass。
- **因果診断**：`overlap_metrics.json` の `overlap>=0.6` 等をCIで閾値判定。
- **Postgres整合**：`select count(*) from causal_ready` が Parquet最終行数と一致。
- **負荷の式（Wolfram Oneで概算）**：読み取り時間 $T \approx \frac{\text{サイズ}}{\text{スループット}}$。
  Parquet圧縮率を $r = \frac{|CSV|}{|Parquet|}$ とすれば $T_{\text{Parquet}} \approx \frac{T_{\text{CSV}}}{r}$。
  典型 $r \in [3, 10]$ → I/O時間が **1/3〜1/10** に短縮（手元I/Oがボトルネックな個人商用で効く）。

What this does（この変更の要旨）

- **Goal**：**多形式入力→契約検証→Parquet正規化→dbt変換→因果診断→Postgres配布**をワンコマンド化。
- **Inputs**：`ciq/data/raw/*.(csv|tsv|jsonl|xlsx|parquet|feather){,gz,bz2}`。
- **Outputs/Side-effects**：`staged.parquet`、`causal_ready.parquet`、`overlap_metrics.json`、**Postgresテーブル** `causal_ready`。
- **Why this approach**：**計算は列指向、配布はPostgres**で役割分担し、**最小差分で商用の堅牢性**を得る。
- **How to run/verify**：`make all_plus_pg`。dbt/pytest/Overlap閾値、PGの件数一致を確認。

よくある反論への先回り

- **「Postgresだけでやりたい」**：可能だが、列指向ETLに不利。個人商用では**開発速度/保守性/将来移行**の観点から非推奨。
- **「何でもアップさせたい」**：サポート外形式が増えるほど**脆弱性/事故/工数**が指数的に上がる。**ホワイトリスト＆契約**が現実解。

Expert insight (Google/Meta/NASAレベル)

Control-plane と Data-plane を分離せよ。PostgreSQLは**ユーザ・認可・監査・配布**の“制御面”に集中させ、**列指向の前処理/分析はParquet系**に固定する。これで**機能の衝突を避け、運用SLAと性能SLOを別々に最適化**できる（小規模でも効く設計原則）。

PostgerSQL・DuckDBの使用判断を自動で実施は可能？

思考時間: 47s >

結論（3行サマリ）

1. **可能**：簡易な“クエリ/状況に応じたエンジン自動ルーティング”を実装できます（DuckDB⇄PostgreSQL）。
2. **方針**：**データの重力（Parquet/列指向はDuckDB、OLTP/配布はPostgres）**＋**文の種別（SELECT/DDL/DML）**＋**実行文脈（API/バッチ）**で決定。
3. **安全策**：**書き込み系はPGへ固定しFail-close**、**読み取りのみ限定でフォールバック**（DuckDB→PG）を許可。

目的 / Non-goals

- **Purpose**：クエリ/入力規模/実行文脈から**DuckDB or PostgreSQL**を自動選択し、個人商用プロダクトで**速度×安全性×保守性**を両立。
- **Non-goals**：
  - 完全なコストベース・オプティマイザの再発明はしない（ヒューリスティック＋軽量見積もり）。
  - 分散実行（Spark等）までは今回含めない（将来拡張フックのみ）。

理念・代替案・トレードオフ

- **採用：ヒューリスティック・ルーティング**（軽量）
  - ルール：
    - **DDL/DML/トランザクション**（`INSERT/UPDATE/DELETE/CREATE/ALTER`）→ **Postgres固定**。
    - **SELECTでParquet/CSV/Feather/JSONLを直接参照**（`read_parquet` など）→ **DuckDB**。
    - **API/配布スキーマ（`public`,`api`）中心の軽集計** → **Postgres**。
    - **入力サイズ（合計MB/行数）**がしきい値超え → **DuckDB**。
    - `/* engine:pg|duckdb */` の**ヒントコメント**で上書き可。
  - 長所：実装が小さく、**最小差分で導入**できる。
  - 短所：厳密最適ではない（ただし個人商用規模では十分実用）。
- **代替：常にDuckDB→Postgresへ配布**
  - 長所：設計が単純。
  - 短所：API経由の小クエリもDuckDB経由になり、**接続/起動コスト**や**権限制御**が煩雑化。
- **代替：常にPostgres**
  - 長所：単一運用。
  - 短所：列指向I/Oで不利、**大きめSELECTが遅く**なる。

境界（I/O, 失敗時, 拡張性）

- **入力**：SQL文字列 or .sql ファイル、任意のParquet/CSV等（`ciq/data/...`）。
- **出力**：SELECTはDataFrame/CSV、DMLは行数。実行メタ（どのエンジンを使ったか）をログ出力。
- **失敗**：
  - **書き込み系はフォールバック禁止**（副作用二重化を避ける）。
  - **読み取りのみ** DuckDB失敗→Postgresへ**限定フォールバック**（安全側に倒す）。
- **拡張**：しきい値/スキーマ/文脈は `router.yml` で管理。将来Spark/BigQuery/Snowflakeを“候補”に追加可能。

実装（最小・監査可能な差分）

依存

```
source .venv/bin/activate
pip install psycopg2-binary sqlparse
```

設定 `ciq/config/router.yml`

```
defaults:
  large_select_mb: 50        # これ以上の入力サイズはDuckDB
  api_schemas: ["public", "api", "auth"]
  pg_write_schemas: ["public", "api"]
rules:
  prefer_duckdb_if_parquet: true
  allow_select_fallback: true
```

エンジン薄ラッパ `ciq/lib/engines.py`

```
# ciq/lib/engines.py
from __future__ import annotations
import time, contextlib
import duckdb, pandas as pd, psycopg2, psycopg2.extras as _e

class DuckEngine:
    def __init__(self, db_path="ciq/warehouse/warehouse.duckdb"):
        self.db_path = db_path
    @contextlib.contextmanager
    def conn(self):
        con = duckdb.connect(self.db_path)
        try: yield con
        finally: con.close()
    def execute(self, sql: str) -> dict:
        t0=time.time()
        with self.conn() as con:
            cur = con.execute(sql)
            try:
                df = cur.fetch_df()
                return {"engine":"duckdb","elapsed":time.time()-t0,"df":df}
            except duckdb.IOException:
                return {"engine":"duckdb","elapsed":time.time()-t0,"rowcount":cur.rowcount}

class PGEngine:
    def __init__(self, dsn: str):
        self.dsn = dsn
    @contextlib.contextmanager
    def conn(self):
        conn = psycopg2.connect(self.dsn)
        try: yield conn
        finally: conn.close()
    def execute(self, sql: str) -> dict:
        t0=time.time()
        with self.conn() as c:
            # SELECT判定
            if sql.lstrip().lower().startswith("select"):
                df = pd.read_sql(sql, c)
                return {"engine":"pg","elapsed":time.time()-t0,"df":df}
            else:
                with c.cursor(cursor_factory=_e.DictCursor) as cur:
                    cur.execute(sql); c.commit()
                    return {"engine":"pg","elapsed":time.time()-t0,"rowcount":cur.rowcount}
```

ルータ本体 `ciq/lib/router.py`

```
# ciq/lib/router.py
from __future__ import annotations
import re, os, glob, yaml, sqlparse
from pathlib import Path
from .engines import DuckEngine, PGEngine

HINT_RE = re.compile(r"/\*\s*engine\s*:\s*(pg|duckdb)\s*\*/", re.I)
READ_PARQUET_RE = re.compile(r"read_parquet\s*\(\s*'([^']+)'", re.I)
FROM_TOKEN_RE = re.compile(r"\bfrom\s+([a-zA-Z0-9_\.\"']+)", re.I)

class EngineRouter:
    def __init__(self, pg_dsn: str, duck_path="ciq/warehouse/warehouse.duckdb",
                 cfg_path="ciq/config/router.yml"):
        self.pg = PGEngine(pg_dsn)
        self.duck = DuckEngine(duck_path)
        self.cfg = yaml.safe_load(Path(cfg_path).read_text())
    def _stmt_type(self, sql:str)->str:
        parsed = sqlparse.parse(sql)
        if not parsed: return "unknown"
        t = parsed[0].get_type().lower()  # 'SELECT','INSERT',...
        return t
```

```
    def _total_input_mb(self, sql:str)->float:
        mb=0.0
        for m in READ_PARQUET_RE.finditer(sql):
            pat = m.group(1)
            for p in glob.glob(pat):
                if os.path.isfile(p):
                    mb += os.path.getsize(p)/1024/1024
        return mb
    def _schemas_in_query(self, sql:str)->set[str]:
        schemas=set()
        for m in FROM_TOKEN_RE.finditer(sql):
            token=m.group(1).strip('"\'')
            if "." in token:
                schemas.add(token.split(".")[0])
        return schemas
    def decide(self, sql:str, context:str="batch")->str:
        # 1) ヒントコメント優先
        hint = HINT_RE.search(sql)
        if hint: return hint.group(1).lower()

        typ = self._stmt_type(sql)
        cfg = self.cfg["defaults"]; rules = self.cfg["rules"]
        api_schemas=set(cfg["api_schemas"])

        # 2) 書き込み/DDLはPG固定
        if typ in {"insert","update","delete","create","alter","drop"}:
            return "pg"

        # 3) コンテキスト: APIはPG優先
        if context=="api":
            return "pg"

        # 4) Parquet直接参照 or 大きめSELECT はDuckDB
        if rules.get("prefer_duckdb_if_parquet") and READ_PARQUET_RE.search(sql):
            return "duckdb"
        if typ=="select" and self._total_input_mb(sql) >= cfg["large_select_mb"]:
            return "duckdb"

        # 5) API系スキーマ参照ならPG
        if self._schemas_in_query(sql) & api_schemas:
            return "pg"

        # 6) デフォルト
        return "duckdb" if typ=="select" else "pg"

    def execute(self, sql:str, context:str="batch") -> dict:
        choice = self.decide(sql, context=context)
        try:
            if choice=="duckdb":
                return self.duck.execute(sql) | {"chosen": "duckdb"}
            else:
                return self.pg.execute(sql) | {"chosen": "pg"}
        except Exception as e:
            # 読み取りのみフォールバック許容
            if self.cfg["rules"].get("allow_select_fallback", True) and \
               self._stmt_type(sql)=="select":
                other = "pg" if choice=="duckdb" else "duckdb"
                try:
                    if other=="pg":
                        return self.pg.execute(sql) | {"chosen": choice, "fallback":"pg", "error":str(e)}
                    else:
                        return self.duck.execute(sql) | {"chosen": choice, "fallback":"duckdb", "error":str(e)}
                except Exception as e2:
                    raise RuntimeError(f"both engines failed: {e} / {e2}")
            raise
```

CLIラッパ `ciq/scripts/run_sql.py`

```python
# ciq/scripts/run_sql.py
import argparse, sys, pandas as pd
from pathlib import Path
from lib.router import EngineRouter

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--query", help="SQL text")
    ap.add_argument("--file", help=".sql file")
    ap.add_argument("--context", default="batch", choices=["batch","api"])
    ap.add_argument("--engine", default="auto", choices=["auto","pg","duckdb"])
    args = ap.parse_args()

    sql = args.query or Path(args.file).read_text()
    r = EngineRouter(pg_dsn=os.environ.get("PG_DSN","postgresql://localhost/ciqdb"))
    if args.engine!="auto":
        # ヒントで強制
        sql = f"/* engine:{args.engine} */\n{sql}"
    res = r.execute(sql, context=args.context)
    print({"engine":res.get("engine"), "chosen":res.get("chosen"),
           "fallback":res.get("fallback"), "elapsed":res.get("elapsed")})
    if "df" in res:
        df: pd.DataFrame = res["df"]
        print(df.head(20).to_string(index=False))
    else:
        print("rowcount:", res.get("rowcount"))

if __name__ == "__main__":
    import os; main()
```

テスト `ciq/tests/test_router.py`

```python
# ciq/tests/test_router.py
from lib.router import EngineRouter

def test_insert_goes_pg(monkeypatch, tmp_path):
    r = EngineRouter(pg_dsn="postgresql://localhost/ciqdb")
    assert r.decide("INSERT INTO public.t(a) VALUES(1)") == "pg"

def test_parquet_select_duckdb(tmp_path):
    # 疑似パスだけで判定（read_parquet検出）
    r = EngineRouter(pg_dsn="postgresql://localhost/ciqdb")
    assert r.decide("select * from read_parquet('ciq/data/staged/*.parquet')") == "duckdb"

def test_hint_overrides():
    r = EngineRouter(pg_dsn="postgresql://localhost/ciqdb")
    assert r.decide("/* engine:pg */ select 1") == "pg"
```

Makefile 追記

```
cat >> Makefile <<'MK'

router_deps:
    $(ACT); pip install psycopg2-binary sqlparse

router_test:
    $(ACT); pytest -q ciq/tests/test_router.py

run_sql_auto:
    $(ACT); python ciq/scripts/run_sql.py --file sample.sql --context batch

MK
```

実行例（手順）

```
# 依存導入
make router_deps

# 例：Parquet由来の重SELECT（自動でDuckDB）
echo "select count(*) from read_parquet('ciq/data/staged/*.parquet');" > sample.sql
make run_sql_auto

# 例：API用途の小SELECTをPGに強制（ヒント）
python ciq/scripts/run_sql.py --query "/* engine:pg */ select count(*) from public.causal_ready" --context api

# テスト
make router_test
```

検証（pytest/メトリクス）

- `pytest ciq/tests/test_router.py`：基本ルール（書き込み→PG、Parquet参照→DuckDB、ヒント優先）を検証。
- 実計測：`elapsed` をログ収集し、**サイズ vs エンジン**の実効時間を記録。
- しきい値調整：`router.yml` の `large_select_mb` を **P90実行時間が最小**になるようチューニング。

簡易コスト式（設計判断の裏付け）

読み取りコストを $T \approx T_0 + \frac{B}{\Theta}$ とし、DuckDBを列指向・圧縮率 r とすると、
同一データに対し $T_{\text{Duck}} \approx T_0^{\text{Duck}} + \frac{B/r}{\Theta_{\text{Duck}}}$、
$T_{\text{PG}} \approx T_0^{\text{PG}} + \frac{B}{\Theta_{\text{PG}}}$。
典型的に $r \in [3, 10]$ かつ列指向の $\Theta_{\text{Duck}} \geq \Theta_{\text{PG}}$ → **大きめSELECTはDuckDBが有利**。
（式自体は簡略化。ヒューリスティックの根拠として閾値設計に利用。）

What this does（要旨）

- **Goal**：**DuckDB/PGの自動選択**により、バッチ分析は速く、API/書込は安全に運用。
- **Inputs**：任意のSQL（`--query` or `--file`）、実行文脈（`--context api|batch`）、設定 `router.yml`。
- **Outputs/Side-effects**：どのエンジンで実行したかのメタ＋結果（DF/行数）。書込は**PG固定で副作用の二重化を回避**。
- **Why this approach**：**最小実装**で“データの重力”に沿った実行先を自動化し、**保守容易＆安全側**に倒す。
- **How to run/verify**：`make router_deps && make router_test && make run_sql_auto`。実行ログの `engine/chosen/fallback/elapsed` を監査。

よくある質問への先回り

- **Q: 自動フォールバックでデータ壊れない？**
  A: **読み取り限定**に制限。DML/DDLはPG固定＆フォールバック禁止で**Fail-close**。
- **Q: PGテーブルとParquetのJOINは？**
  A: ルータの責務外。**ETLでどちらかに寄せてから**JOIN（推奨はDuckDB側へ一時取り込み）。
- **Q: “最適”と断言できる？**
  A: いいえ。**実測で閾値を継続チューニング**（P90/スループット/同時実行）し、`router.yml` に反映する運用が前提。

Expert insight (Google/Meta/NASAレベル)

“自動ルーティングは機能ではなく安全契約**から設計せよ**。**
書き込み/権限/監査が絡む経路は**常にPGへ固定**し、**自動化は**“読み取りの最適化”に限定**する。副作用がない範囲でのみフォールバックを許可すると、**SLA（正しさ）とSLO（速さ）を分離**でき、バグ時の被害半径を最小化できる。