

User:

如图这是我现在单独存放的lob数据，列名之前和你说过ap1,bp1...，当然是不带成交数据的,但是现在有个问题，目前数据按照日期分类的，但是有些不同日期文件夹里的symbol可能不一样，比如某一天没有ETH的数据，而且有些天的数据不是完全的，可能有些空缺，比如8点到9点的数据没有，请你给我设计一套处理方案以便于我后面的数据计算，可能会用到回测和因子计算

ChatGPT:

好的，我给你一套面向回测/因子计算的 LOB 数据处理方案（含可直接跑的 Polars 代码模板）。目标是：在“按日期分目录、部分 symbol 缺失、日内有时间段缺口”的前提下，产出**统一时间栅格**、**带可用性掩码(mask)**与**质量评分**的面板数据，并可稳定复用与缓存。

处理思路（浓缩版）

1. 统一命名 & 目录结构（不改你现有存量也行）

- 继续保留 YYYY-MM-DD/ 的原目录，但构建一份 **manifest（清单）**：每个 {date, symbol} 的文件路径、起止时间、行数、质量分、覆盖率等汇总到 parquet。
- 额外建议（可选）：生成一个按 symbol/date= 的**二级索引**（软链接或映射表），方便按 symbol 批处理与回测预热。

2. 质量校验（per file）

- 时间戳单调、去重；
- 盘口是否交叉（bp1 >= ap1）与异常价比例；
- 心跳统计：相邻 Δt 分布、极端长间隔；
- 列完整性（ap1,av1,...,bp1,bv1,...,T(ms)）。

3. 统一时间栅格 + 掩码

- 按需求生成心跳（如 1s 或 200ms）：用 **asof 回填**（上一个快照）到栅格；
- 计算 **staleness**（栅格时刻与上一个真实快照的时间差），并生成
 - mask_live = (staleness_ms <= max_age_ms)（小缺口“拿前值”但标记为 live），
 - 大缺口直接 mask_live=False；
- 产出 segment_id（遇到大缺口就换段），滚动特征只在段内计算，避免跨缺口污染。

4. 覆盖度与质量分

- 以目标栅格为基准计算 coverage_ratio、crossed_quote_ratio、mean_spread(bps)、median_dt 等，写入 manifest。

- 后续可基于阈值（如 coverage>=0.9、crossed<0.1%）自动筛选可交易宇宙（rolling tradable universe）。

5. 缺 symbol / 缺时段的处理原则

- 横截面因子：任何时刻仅在 mask_live=True 的 symbols 上计算与排序（自然忽略缺失）。
- 持仓遇到缺口：回测规则明确化：
 - 小缺口（≤ max_age_ms）：允许继续持仓，用“陈值+老化惩罚”。
 - 大缺口：在缺口结束后的第一笔可交易价按“mid±1/2spread+滑点惩罚”结算或减仓。
- 滚动窗口：仅在同一 segment_id 内滚（如 group_by("symbol","segment_id") 再做 rolling）。

6. 缓存

- 对每个 {date, symbol} 输出 已栅格化的 Parquet（含 mask_live、staleness_ms、segment_id、mid、spread 等），二次使用时直接 scan_parquet，极快。

可直接跑的最小实现（Polars）

说明：自动扫描 root_dir/YYYY-MM-DD/*，尝试读 csv/feather/parquet（自行按实际改 read_one()），生成 manifest，并输出按天按币种的栅格化 parquet，最后提供加载面板与示例因子计算接口。

```
python
# lob_pipeline.py
from __future__ import annotations
import re
from pathlib import Path
from typing import Dict, List, Tuple, Optional
import polars as pl

# ===== 配置 =====
TIME_UNIT = "ms" # 统一时间单位
DEFAULT_FREQ = "1s" # 统一栅格
MAX_STALENESS_MS = 3000 # 小缺口允许的最大“陈旧”时长
MIN_SEG_GAP_MS = 5000 # 段切分阈值：超过视为大缺口 -> 新 segment
MANIFEST_PATH = "manifest.parquet"
GRID_CACHE_DIR = "grid_parquet" # 栅格化后缓存输出目录

# 典型 L2 列；按你实际列名微调
ASK_PX = [f"ap{i}" for i in range(1, 21)]
ASK_VOL = [f"av{i}" for i in range(1, 21)]
BID_PX = [f"bp{i}" for i in range(1, 21)]
BID_VOL = [f"bv{i}" for i in range(1, 21)]
LOB_COLS = ASK_PX + ASK_VOL + BID_PX + BID_VOL
TS_COL = "T" # 原始毫秒时间戳
```

```
# ===== 工具 =====
def parse_symbol_and_date(p: Path) -> Tuple[str, str]:
    # 目录名就是 YYYY-MM-DD
    date_str = p.parent.name
    # 文件名形如 "AAVE-2025-05-06-..." -> 取首段为 symbol
    m = re.match(r"([A-Za-z0-9_]+)-", p.name)
    sym = m.group(1) if m else p.stem.split("-")[0]
    return sym, date_str

def read_one(p: Path) -> pl.DataFrame:
    # 自行按实际格式改: 示例兼容 csv/parquet/feather
    suffix = p.suffix.lower()
    if suffix == ".parquet":
        df = pl.read_parquet(p)
    elif suffix in (".feather", ".ft"):
        df = pl.read_ipc(p)
    else:
        # 假定 csv, 含表头
        df = pl.read_csv(p, infer_schema_length=1000)
    if TS_COL not in df.columns:
        # 兜底兼容: 尝试 'ts'/'timestamp' 列
        for alt in ["ts", "timestamp"]:
            if alt in df.columns:
                df = df.rename({alt: TS_COL})
                break
    return df

def basic_clean(df: pl.DataFrame) -> pl.DataFrame:
    # 仅保留 LOB 列与 T
    keep = [c for c in df.columns if c in LOB_COLS] + [TS_COL]
    df = df.select(keep).drop_nulls(subset=[TS_COL])
    # 确保类型正确
    df = df.with_columns(pl.col(TS_COL).cast(pl.Int64))
    # 排序&按时间去重 (同 T 取最后一条)
    df = df.sort(TS_COL).unique(subset=[TS_COL], keep="last")
    # 加 datetime 列
    df = df.with_columns(
        pl.from_epoch(pl.col(TS_COL), unit=TIME_UNIT).alias("ts")
    )
    return df

def quality_stats(df: pl.DataFrame) -> Dict:
    if df.is_empty():
        return {
            "rows": 0,
            "t_min": None,
            "t_max": None,
            "median_dt_ms": None,
            "crossed_ratio": None,
            "mean_spread_bps": None,
```

```
}
# 顶档
has_best = ("ap1" in df.columns) and ("bp1" in df.columns)
crossed = pl.lit(None)
spread_bps = pl.lit(None)
if has_best:
    crossed = (pl.col("bp1") >= pl.col("ap1")).mean() # 交叉比例
    spread_bps = ( (pl.col("ap1") - pl.col("bp1")) / pl.col("bp1").abs() * 1e4 ).mean()

dt = pl.col(TS_COL).diff().alias("dt")
q = df.select(
    pl.len().alias("rows"),
    pl.col(TS_COL).min().alias("t_min"),
    pl.col(TS_COL).max().alias("t_max"),
    dt
)
med_dt = q["dt"].drop_nulls().median() if q["rows"][0] > 1 else None

return {
    "rows": int(q["rows"][0]),
    "t_min": int(q["t_min"][0]) if q["rows"][0] > 0 else None,
    "t_max": int(q["t_max"][0]) if q["rows"][0] > 0 else None,
    "median_dt_ms": int(med_dt) if med_dt is not None else None,
    "crossed_ratio": float(df.select(crossed)["bp1"].item()) if has_best else None,
    "mean_spread_bps": float(df.select(spread_bps)["ap1"].item()) if has_best else None,
}

def resample_to_grid(
    df: pl.DataFrame,
    every: str = DEFAULT_FREQ,
    max_staleness_ms: int = MAX_STALENESS_MS,
    min_seg_gap_ms: int = MIN_SEG_GAP_MS,
) -> pl.DataFrame:
    if df.is_empty():
        return df

    t0 = df["ts"].min()
    t1 = df["ts"].max()
    grid = pl.DataFrame({
        "ts": pl.datetime_range(start=t0, end=t1, interval=every, time_unit=TIME_UNIT, eager=True)
    })
    # 为 asof 准备: 右表的匹配键改名为 last_ts
    right = df.rename({"ts": "last_ts"})

    joined = grid.join_asof(
        right,
        left_on="ts", right_on="last_ts",
        strategy="backward"
    )

    # 陈旧度 (可能为 null: 表示此前没有任何观测)
```

```
staleness = (pl.col("ts").cast(pl.Int64) - pl.col("last_ts").cast(pl.Int64)).alias("staleness_ms")
joined = joined.with_columns(staleness)

# 掩码: 小缺口以内视为 live
joined = joined.with_columns(
    (pl.col("staleness_ms").is_not_null() & (pl.col("staleness_ms") <= max_staleness_ms)).alias("mask_live")
)

# 段切分: 超过 min_seg_gap_ms 视为断档
gap_flag = (pl.col("staleness_ms").is_null() | (pl.col("staleness_ms") > min_seg_gap_ms))
joined = joined.with_columns(
    gap_flag.fill_null(True).cast(pl.Int8).diff().fill_null(0).ne(0).cumsum().alias("segment_id")
)

# 常用派生: mid / spread
if "ap1" in joined.columns and "bp1" in joined.columns:
    joined = joined.with_columns([
        ((pl.col("ap1") + pl.col("bp1")) * 0.5).alias("mid"),
        (pl.col("ap1") - pl.col("bp1")).alias("spread")
    ])

return joined

def grid_quality(joined: pl.DataFrame) -> Dict:
    if joined.is_empty():
        return {"grid_rows": 0, "coverage_ratio": 0.0}
    n = joined.height
    cov = float(joined.select(pl.col("mask_live").fill_null(False).mean()).item())
    return {"grid_rows": n, "coverage_ratio": cov}

# ===== 主流程 =====
def build_manifest(root_dir: str, every: str = DEFAULT_FREQ) -> pl.DataFrame:
    root = Path(root_dir)
    rows = []
    for date_dir in sorted([p for p in root.iterdir() if p.is_dir()]):
        date_str = date_dir.name
        for f in sorted(date_dir.iterdir()):
            if f.is_dir():
                continue
            sym, _ = parse_symbol_and_date(f)
            try:
                raw = read_one(f)
                raw = basic_clean(raw)
                q = quality_stats(raw)
                grd = resample_to_grid(raw, every=every)
                gq = grid_quality(grd)
                rows.append({
                    "date": date_str,
                    "symbol": sym,
                    "path": str(f),
                    **q, **gq
                })
            except:
```

```

    })
    except Exception as e:
        rows.append({
            "date": date_str, "symbol": sym, "path": str(f),
            "rows": 0, "t_min": None, "t_max": None, "median_dt_ms": None,
            "crossed_ratio": None, "mean_spread_bps": None,
            "grid_rows": 0, "coverage_ratio": 0.0,
            "error": str(e)
        })
    mf = pl.DataFrame(rows)
    mf.write_parquet(MANIFEST_PATH)
    return mf

def materialize_grid_parquet(
    root_dir: str,
    out_dir: str = GRID_CACHE_DIR,
    every: str = DEFAULT_FREQ,
    max_staleness_ms: int = MAX_STALENESS_MS,
    min_seg_gap_ms: int = MIN_SEG_GAP_MS,
    overwrite: bool = False
):
    root = Path(root_dir)
    out = Path(out_dir)
    out.mkdir(parents=True, exist_ok=True)

    for date_dir in sorted([p for p in root.iterdir() if p.is_dir()]):
        date_str = date_dir.name
        (out / date_str).mkdir(exist_ok=True)
        for f in sorted(date_dir.iterdir()):
            if f.is_dir():
                continue
            sym, _ = parse_symbol_and_date(f)
            out_path = out / date_str / f"{sym}.parquet"
            if out_path.exists() and not overwrite:
                continue
            raw = basic_clean(read_one(f))
            joined = resample_to_grid(raw, every=every,
                                     max_staleness_ms=max_staleness_ms,
                                     min_seg_gap_ms=min_seg_gap_ms)
            # 附带 symbol, 方便后续 concat
            joined = joined.with_columns(pl.lit(sym).alias("symbol"))
            joined.write_parquet(out_path)

def select_tradable_universe(
    manifest_path: str = MANIFEST_PATH,
    min_coverage: float = 0.90,
    min_rows: int = 3600 # 对应 1Hz 栅格下 ≥1小时
) -> pl.DataFrame:
    mf = pl.read_parquet(manifest_path)
    return (
        mf.filter(

```

```

        (pl.col("coverage_ratio") >= min_coverage) &
        (pl.col("grid_rows") >= min_rows)
    )
    .select(["date", "symbol", "coverage_ratio", "grid_rows", "mean_spread_bps", "crossed_ratio"])
    .sort(["date", "symbol"])
)

def load_panel(
    cache_dir: str,
    start_date: str,
    end_date: str,
    symbols: Optional[List[str]] = None
) -> pl.DataFrame:
    root = Path(cache_dir)
    dates = [p.name for p in sorted(root.iterdir()) if p.is_dir() and (start_date <= p.name <= end_date)]
    dfs = []
    for d in dates:
        day_dir = root / d
        files = list(day_dir.glob("*.parquet"))
        for fp in files:
            sym = fp.stem
            if (symbols is not None) and (sym not in symbols):
                continue
            df = pl.read_parquet(fp)
            dfs.append(df.select(["ts", "symbol", "mid", "spread", "mask_live", "staleness_ms", "segment_id"] + [c for c in LOB_COLS if c in df.columns]))
    return pl.concat(dfs, how="diagonal_relaxed") if dfs else pl.DataFrame()

# ===== 示例：横截面因子（顶档不平衡） =====
def top1_imbalance(df: pl.DataFrame) -> pl.DataFrame:
    # 仅在 mask_live 的时刻计算
    expr = None
    if all(c in df.columns for c in ("av1", "bv1")):
        expr = (pl.col("bv1") - pl.col("av1")) / (pl.col("bv1") + pl.col("av1")).clip_min(1e-12).alias("imb1")
    else:
        return df
    return df.with_columns(expr).with_columns(
        pl.when(pl.col("mask_live")).then(pl.col("imb1")).otherwise(None).alias("imb1")
    )

if __name__ == "__main__":
    ROOT = "/path/to/your/LOB_root" # 改成你的根目录
    # 1) 构建清单
    mf = build_manifest(ROOT, every=DEFAULT_FREQ)
    print(mf.head())

    # 2) 栅格化并缓存
    materialize_grid_parquet(ROOT, out_dir=GRID_CACHE_DIR, every=DEFAULT_FREQ)

    # 3) 选 tradable 宇宙
    tradable = select_tradable_universe(MANIFEST_PATH, min_coverage=0.9, min_rows=12*3600)
    print(tradable.head())

```

```
# 4) 载入一段时间的面板并计算示例因子
panel = load_panel(GRID_CACHE_DIR, "2025-05-01", "2025-05-07", symbols=None)
panel = top1_imbalance(panel)
print(panel.head())
```

回测与因子计算的关键约定

- **时间对齐**：所有计算以 ts 栅格为基准；事件驱动策略也先落到栅格（更可控）。
- **掩码使用**：mask_live=True 时才参与排名/下单/更新仓位；False 时窗口断开（靠 segment_id 控制 rolling）。
- **滑点/成本**：撮合价建议 $exec_price = mid \pm 0.5 * spread + slippage_penalty$ ；penalty 可按 staleness_ms 与 spread 的函数动态加大。
- **宇宙选择**：用 manifest 的 coverage_ratio + crossed_ratio + mean_spread_bps 做多维过滤，减少脏数据对回测的影响。
- **跨日滚动**：计算长窗口（如 30 分钟）时，拼接相邻日期的尾部缓存再 rolling；但仍受 segment_id 限制，避免跨大缺口。

你可以直接这样用

1. 把脚本里的 ROOT 改成你的根目录，跑一遍生成 manifest.parquet 与 grid_parquet/YYYY-MM-DD/SYMBOL.parquet。
2. 用 select_tradable_universe 选出可用宇宙（比如覆盖≥0.9、有效时长≥12h）。
3. 用 load_panel 拉时间段的面板，再在此基础上写你的因子/信号/回测逻辑（示例里给了顶档不平衡 imb1）。

如果你希望我把**“持仓遇缺口如何精确结算”、“跨日滚动的尾部拼接”、或“以 LazyFrame 全量并行化 + Hive 分区”**继续补成生产级版本，我可以直接把这些函数再扩一版给你。

User:

好的，那么如图，这是我单独保存的aggtrades数据，这些数据是我从binance官网下载的，应该是全的，而且单个文件是一个symbol一个月的数据，那么如何将aggtrades数据和上面的lob数据处理流程结合起来呢？

ChatGPT:

很棒，这样刚好把“全量、但按月保存的 aggTrades”和“不一定全、按日保存的 LOB”拼到同一根时间轴上。下面给你一套三层管线（设计要点 + 可直接跑的 Polars 代码模板），输出可复用的日级缓存，用于回测/因子。

设计要点（核心规则）

1. 统一时间轴（UTC）与栅格

- 选择一个频率（常用 1s 或 200ms），LOB 用 *asof* 回填（产生 mask_live/staleness_ms/segment_id），aggTrades 用 时间分箱（group_by_dynamic(label="right", closed="right")）。
- 两者最后都产出列 ts（Datetime[ms]），按 ts 左连（以 LOB 为主键）。

2. aggTrades → 栅格化的成交特征（每个 bin）

- n_trades, vol, vol_buy, vol_sell, dollar_vol, vwap, last_price, ret
- sign = +1 表示买方主动（Binance 的 is_buyer_maker==False），-1 表示卖方主动。
- of_buy_ratio = vol_buy / (vol_buy + vol_sell)、signed_dv = $\sum \text{sign} * \text{price} * \text{qty}$ 。
- 生成 mask_trade = (n_trades>0)，并统计日内 trade_coverage（用于质检和宇宙筛选）。

3. 合并策略

- 主表：LOB 栅格（有 mask_live/segment_id），把 trade bins left join 到同一 ts。
- 回测/因子计算仅在 mask_live=True 的时刻使用LOB相关特征，trade特征可在 mask_trade=False 时补零（代表该秒无成交）。
- 滚动窗口/累计量一律在同一 segment_id 内计算，避免跨大缺口。

4. 质量/覆盖筛选

- LOB: coverage_ratio≥阈值、crossed_quote_ratio 限制；
- Trades: trade_coverage≥阈值、median_trade_dt；
- 形成“tradable universe”规则，横截面策略按此筛选。

代码模板（可直接跑）

假设你已经有上一条消息里的 lob_pipeline.py（生成 LOB 栅格缓存）。下面这份把 月度 aggTrades 做成日级栅格缓存，再与 LOB 合并。

```
python
# lob_trades_integration.py
from __future__ import annotations
from pathlib import Path
import re
import polars as pl

# ===== 全局配置（与 LOB 保持一致） =====
TIME_UNIT = "ms"
DEFAULT_FREQ = "1s"
TRADES_GRID_DIR = "trades_grid_parquet" # 输出: trades按日栅格
```

```
MERGED_GRID_DIR = "merged_grid_parquet" # 输出: LOB+Trades合并后
# LOB 缓存目录 (来自上一版管线 materialize_grid_parquet 的输出)
LOB_GRID_DIR = "grid_parquet"

# Binance aggTrades 列 (你给过)
TRADE_COLS = ["agg_trade_id", "price", "quantity", "first_trade_id", "last_trade_id", "transact_time", "is_buyer_maker"]

# ----- 读与清洗 -----
def read_month_file(p: Path) -> pl.DataFrame:
    # feather -> polars ipc
    df = pl.read_ipc(p)
    # 统一列名 (防止大小写)
    ren = {c: c.lower() for c in df.columns}
    df = df.rename(ren)
    # 只保留需要的列
    keep = [c for c in TRADE_COLS if c in df.columns]
    df = df.select(keep)
    # 类型修正
    df = df.with_columns([
        pl.col("transact_time").cast(pl.Int64),
        pl.col("price").cast(pl.Float64),
        pl.col("quantity").cast(pl.Float64),
        pl.col("is_buyer_maker").cast(pl.Boolean)
    ])
    # 时间列
    df = df.with_columns(
        pl.from_epoch(pl.col("transact_time"), unit=TIME_UNIT).alias("ts")
    ).sort("ts")
    return df

def parse_symbol_from_filename(p: Path) -> str:
    # e.g. BTCUSDT-aggTrades-2025-05.feather -> BTCUSDT
    m = re.match(r"([A-Z0-9_]+)-aggTrades-", p.name, flags=re.I)
    return m.group(1).upper() if m else p.stem.split("-")[0].upper()

# ----- 月文件 -> 按日栅格 -----
def month_to_daily_trade_bins(month_file: str, out_dir: str = TRADES_GRID_DIR, every: str = DEFAULT_FREQ):
    out_root = Path(out_dir)
    out_root.mkdir(parents=True, exist_ok=True)

    p = Path(month_file)
    sym = parse_symbol_from_filename(p)
    df = read_month_file(p)

    if df.is_empty():
        return

    # 分日切片
    df = df.with_columns(pl.col("ts").dt.date().alias("d"))
    for day, sub in df.group_by("d", maintain_order=True):
```

```

date_str = str(day)
day_dir = out_root / date_str
day_dir.mkdir(exist_ok=True)
out_path = day_dir / f"{sym}.parquet"
# 按秒分箱 (右闭右标记, 落到bin右端)
bins = (
    sub.group_by_dynamic(index_column="ts", every=every, label="right", closed="right")
        .agg([
            pl.len().alias("n_trades"),
            pl.sum("quantity").alias("vol"),
            pl.sum( pl.when(~pl.col("is_buyer_maker")).then(pl.col("quantity")).otherwise(0.0) ).alias("vol_buy"),
            pl.sum( pl.when(pl.col("is_buyer_maker")).then(pl.col("quantity")).otherwise(0.0) ).alias("vol_sell"),
            pl.sum(pl.col("price") * pl.col("quantity")).alias("dollar_vol"),
            (pl.sum(pl.col("price") * pl.col("quantity")) / pl.sum("quantity")).alias("vwap"),
            pl.col("price").tail(1).alias("last_price")
        ])
        .with_columns([
            pl.col("vol").fill_null(0.0),
            (pl.col("vol_buy") - pl.col("vol_sell")).alias("vol_net"),
            (pl.col("vol_buy") / (pl.col("vol_buy") + pl.col("vol_sell")).clip_min(1e-12)).alias("of_buy_ratio"),
            (pl.col("dollar_vol").fill_null(0.0)).alias("signed_dv") # 注意: 这里是总美元量; 若要“带方向”的美元流见下行
        ])
)

# 方向美元流 (买主动为正, 卖主动为负)
# is_buyer_maker==False => buy-aggr
signed_dv = (
    sub.with_columns(
        (pl.when(~pl.col("is_buyer_maker")).then(1).otherwise(-1) * pl.col("price") * pl.col("quantity")).alias("sdv")
    )
    .group_by_dynamic(index_column="ts", every=every, label="right", closed="right")
    .agg(pl.sum("sdv").alias("signed_dv"))
)

bins = bins.join(signed_dv, on="ts", how="left").with_columns(pl.col("signed_dv").fill_null(0.0))

# 计算 bin 收益 (基于 vwap 或 last_price)
bins = bins.with_columns(
    pl.col("last_price").pct_change().alias("ret_last")
)

# 加 symbol & 掩码
bins = bins.with_columns([
    pl.lit(sym).alias("symbol"),
    (pl.col("n_trades") > 0).alias("mask_trade")
])

# 补齐整天的空bin: 让每秒都有一行 (便于与 LOB 对齐)
t0 = pl.datetime(day.year, day.month, day.day, 0, 0, 0, time_unit=TIME_UNIT)
t1 = pl.datetime(day.year, day.month, day.day, 23, 59, 59, time_unit=TIME_UNIT)
full_grid = pl.DataFrame({"ts": pl.datetime_range(t0, t1, every=every, eager=True, time_unit=TIME_UNIT)})

```

```

bins = full_grid.join(bins, on="ts", how="left").with_columns([
    pl.col("symbol").fill_null(sym),
    pl.col("n_trades").fill_null(0),
    pl.col("vol").fill_null(0.0),
    pl.col("vol_buy").fill_null(0.0),
    pl.col("vol_sell").fill_null(0.0),
    pl.col("vol_net").fill_null(0.0),
    pl.col("dollar_vol").fill_null(0.0),
    pl.col("vwap"),
    pl.col("last_price"),
    pl.col("ret_last").fill_null(0.0),
    pl.col("of_buy_ratio").fill_null(0.5),
    pl.col("signed_dv").fill_null(0.0),
    pl.col("mask_trade").fill_null(False)
])

bins.write_parquet(out_path)

# ----- 批量把 trades 月文件物化为“按日” -----
def materialize_trades_month_dir(trades_dir: str, out_dir: str = TRADES_GRID_DIR, every: str = DEFAULT_FREQ):
    td = Path(trades_dir)
    files = sorted([p for p in td.iterdir() if p.suffix.lower() in (".feather", ".ft", ".ipc", ".parquet")])
    for f in files:
        month_to_daily_trade_bins(str(f), out_dir=out_dir, every=every)

# ----- 合并 LOB 栅格 与 日级 trades 栅格 -----
def merge_lob_and_trades_per_day(date_str: str, symbols: list[str] | None = None,
                                lob_dir: str = LOB_GRID_DIR, trades_dir: str = TRADES_GRID_DIR,
                                out_dir: str = MERGED_GRID_DIR):
    out_day = Path(out_dir) / date_str
    out_day.mkdir(parents=True, exist_ok=True)

    lob_day = Path(lob_dir) / date_str
    tr_day = Path(trades_dir) / date_str
    if not lob_day.exists():
        return

    # 遍历该日的 LOB 符号 (以 LOB 为主, 只有 trades 没有 LOB 的不合并)
    for fp in sorted(lob_day.glob("*.parquet")):
        sym = fp.stem
        if symbols and sym not in symbols:
            continue
        lob = pl.read_parquet(fp) # 含 ts,symbol,mid,spread,mask_live,staleness_ms,segment_id,...
        # 尝试加载 trades 栅格
        tfile = tr_day / f"{sym}.parquet"
        if tfile.exists():
            tb = pl.read_parquet(tfile) # 含 ts,symbol,n_trades,vol,of_buy_ratio,signed_dv,...
        else:
            # 构造空的占位表 (以便左连接后列存在)
            tb = lob.select("ts").with_columns([
                pl.lit(sym).alias("symbol"),

```

```

        pl.lit(0).alias("n_trades"),
        pl.lit(0.0).alias("vol"),
        pl.lit(0.0).alias("vol_buy"),
        pl.lit(0.0).alias("vol_sell"),
        pl.lit(0.0).alias("vol_net"),
        pl.lit(0.0).alias("dollar_vol"),
        pl.lit(None).alias("vwap"),
        pl.lit(None).alias("last_price"),
        pl.lit(0.0).alias("ret_last"),
        pl.lit(0.5).alias("of_buy_ratio"),
        pl.lit(0.0).alias("signed_dv"),
        pl.lit(False).alias("mask_trade"),
    ])

merged = lob.join(tb.drop("symbol"), on="ts", how="left")

# 一些常见派生 (仅在 mask_live 内有效)
merged = merged.with_columns([
    pl.when(pl.col("mask_live")).then( ((pl.col("ap1")+pl.col("bp1"))*0.5) ).otherwise(pl.col("mid")).alias("mid"), # 保底
    pl.when(pl.col("mask_live")).then(pl.col("spread")).otherwise(None).alias("spread")
])

# 保存
(out_day / f"{sym}.parquet").write_bytes(merged.write_parquet(file=None))

# ----- 质量统计: trades 覆盖 -----
def trade_day_coverage(date_str: str, trades_dir: str = TRADES_GRID_DIR) -> pl.DataFrame:
    day_dir = Path(trades_dir) / date_str
    rows = []
    if not day_dir.exists():
        return pl.DataFrame()
    for fp in sorted(day_dir.glob("*.parquet")):
        sym = fp.stem
        df = pl.read_parquet(fp).select(["ts", "n_trades", "mask_trade"])
        cov = float(df.select(pl.col("mask_trade").mean()).item())
        med_dt = (
            df.filter(pl.col("mask_trade"))
            .select(pl.col("ts").cast(pl.Int64).diff().drop_nulls())
            .to_series().median()
        )
        rows.append({"date": date_str, "symbol": sym, "trade_coverage": cov,
                     "median_trade_dt_ms": int(med_dt) if med_dt is not None else None})
    return pl.DataFrame(rows)

if __name__ == "__main__":
    # 1) 先把月度 aggTrades 物化为“按日栅格”
    materialize_trades_month_dir(trades_dir="trades", out_dir=TRADES_GRID_DIR, every=DEFAULT_FREQ)

    # 2) 把某天 LOB 与 trades 合并为统一面板
    merge_lob_and_trades_per_day("2025-05-06", lob_dir=LOB_GRID_DIR, trades_dir=TRADES_GRID_DIR, out_dir=MERGED_GRID_DIR)

```

```
# 3) 看看该日 trades 覆盖度
print(trade_day_coverage("2025-05-06"))
```

回测/因子如何使用（快速指引）

- 读取: merged_grid_parquet/YYYY-MM-DD/SYMBOL.parquet（每秒一行）。
- 可直接用的列：
 - LOB: mid, spread, mask_live, staleness_ms, segment_id, ap1/bp1/av1/bv1...
 - Trades: n_trades, vol, vol_buy/vol_sell/vol_net, of_buy_ratio, signed_dv, vwap, ret_last, mask_trade
- 常见特征：
 - 流量不平衡: of_buy_ratio、vol_net、signed_dv 的滚动均值/标准化（仅在 segment_id 内滚）。
 - Kyle λ （微观冲击）: 在滚动窗口内回归 $\Delta mid \sim signed_dv$, λ 为斜率（建议 WLS, 权重用成交量）。
 - VPIN 近似: 把 signed_dv 的绝对值累和 / 总量, 滚动计算。
 - 交易成本: 下单价可设为 $mid \pm 0.5 * spread + k * spread * (staleness_ms / MAX_STALENESS)$ 。
- 规则：
 - 计算需要 入单价/撤单价 的策略, 必须 mask_live=True; 只用成交流特征的信号, 可在 mask_trade=False 时把流量特征当 0（无成交）。
 - 大缺口跨越: 用 segment_id 断开滚动与持仓逻辑, 缺口后第一笔可交易再续。

小提示

- 时区: Binance 时间戳是 UTC ms。确保 LOB 与 trades 都按 UTC 处理。
- 性能: 以上已使用 Polars 聚合; 如你的月文件特别大, 可改用 scan_ipc() + 懒执行, 并按日期 filter(ts >= d && ts < d+1) 逐日物化。
- 一致性: LOB 的 DEFAULT_FREQ 与 trades 的 every 必须一致, 才能无损对齐。

如果你希望我把 滚动回归的 Kyle λ /VPIN 指标 或 “遇缺口的持仓结算逻辑” 也直接写成函数并接到 MERGED_GRID_DIR, 说一声, 我就把那部分补成可复用模块。

.....