

Adaptive DAGs: Dynamic Query Plan Rewriting and Runtime Task Re-sizing in Airflow

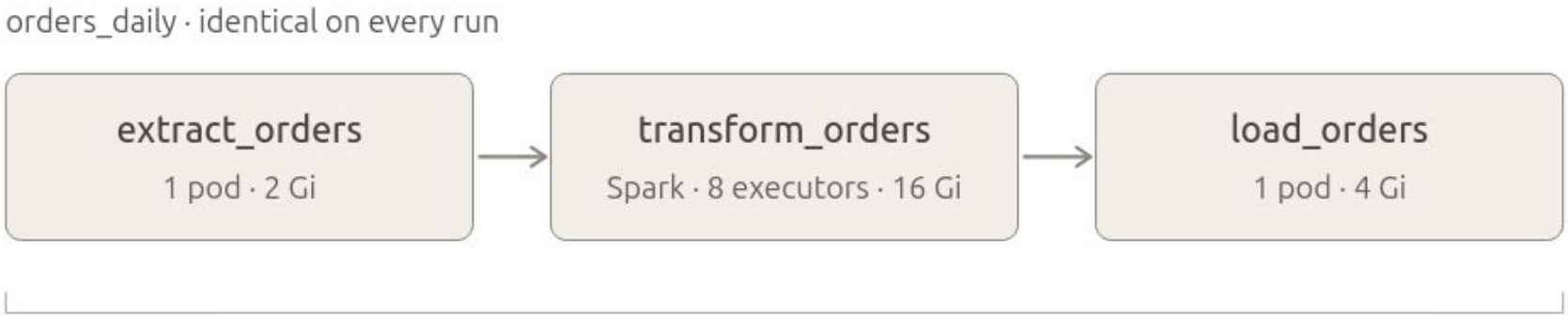
Satej Kumar Sahu

Zalando SE

Disclaimer

The views, opinions, and ideas expressed in this presentation are solely my own and do not reflect the views, positions, or policies of my employer or any organization with which I am affiliated. This content is presented in a personal capacity and should not be interpreted as representing the stance or perspective of my current or past employers.

Traditional DAGs: the plan is fixed before the data exists



All decided when the scheduler parses the DAG file

No task knows how much data it will receive

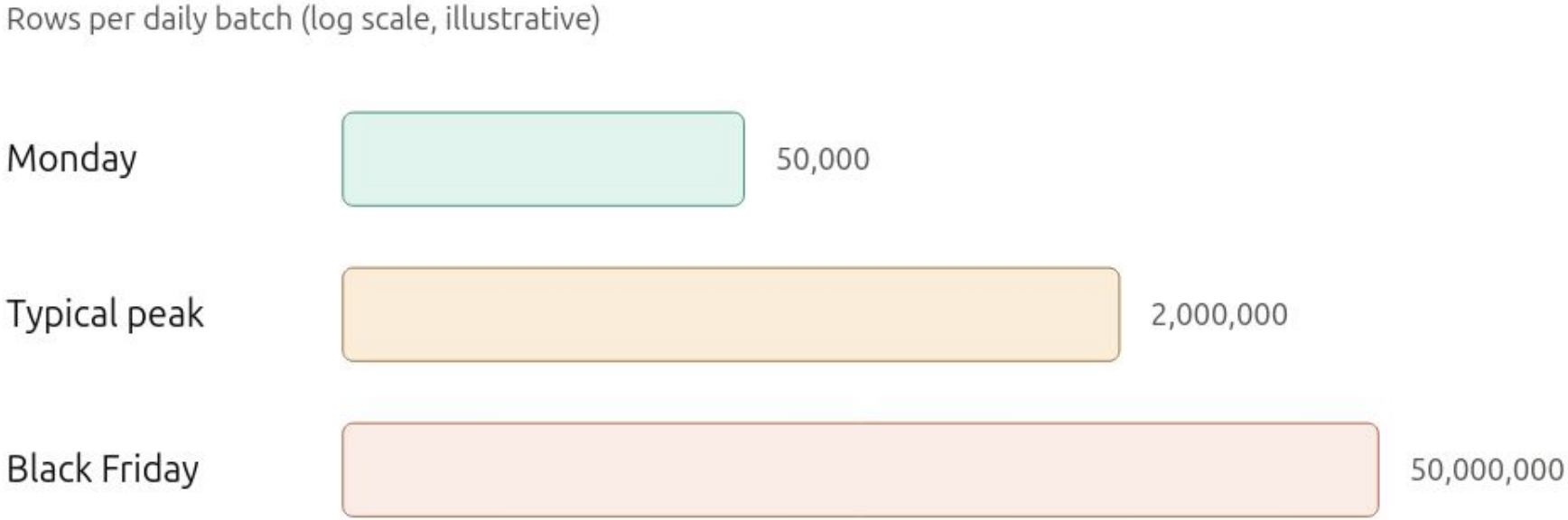
Fixed shape
Same tasks, same edges, same fan-out every run.

Fixed engine
Spark is used even when the batch fits in a laptop.

Fixed size
CPU, memory and parallelism are hardcoded in the operator.

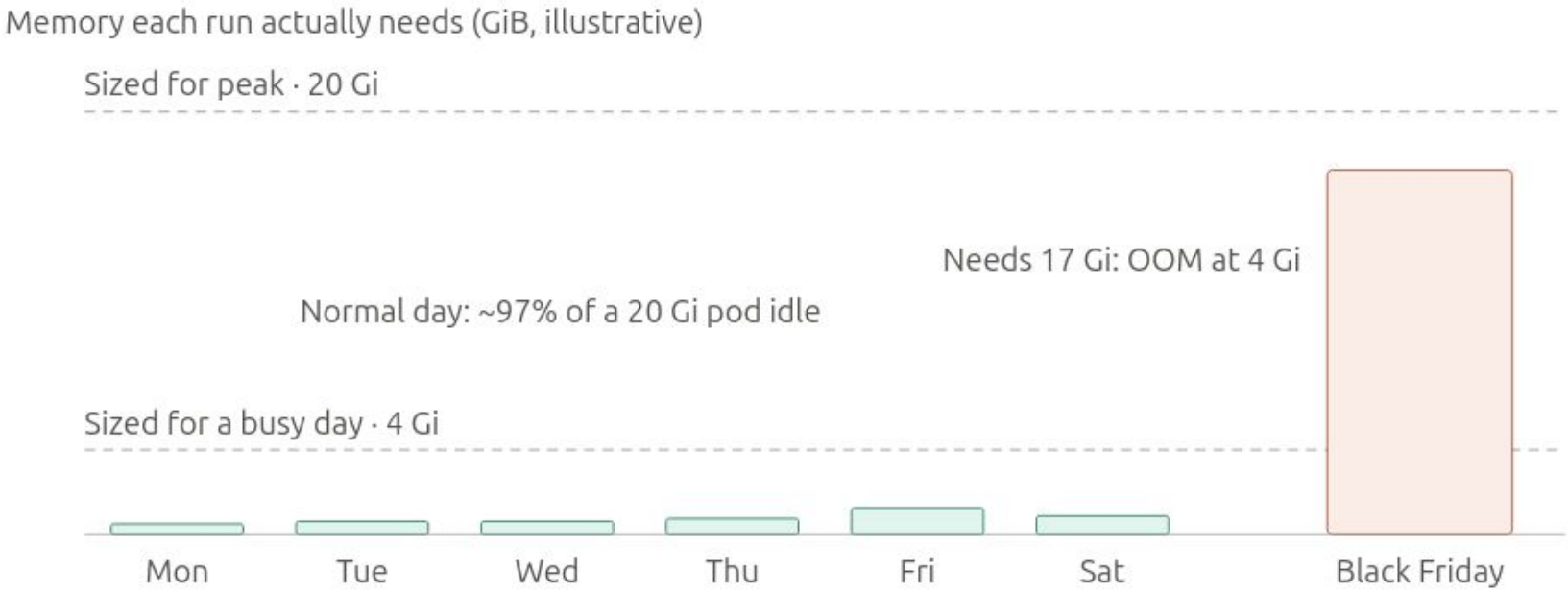
The DAG is a pre-compiled physical plan, and it runs unchanged whether the batch is 50K rows or 50M.

Same DAG, same code, 1000× the data



The logical transformation is unchanged. The physical workload is not.

Static sizing puts cost and reliability in conflict



Size for the peak

Pay for 20 Gi to process megabytes, every day.

Size for the average

OOM on the peak day; identical retries fail again.

Skew breaks both

One hot partition OOMs a single task while others idle.

Plan time vs run time

Static DAG



Adaptive DAG



Move the sizing decision to the latest point where cheap, accurate information exists — just before the heavy stage.

Adaptive doesn't mean editing the running DAG

Fixed at parse time

- ✓ Task IDs and permitted branches
- ✓ Dependencies and output contracts
- ✓ Reusable task templates
- ✓ Predeclared resource tiers

Decided at run time

- ✓ `@task.branch` picks an eligible path
- ✓ `.expand_kwargs()` creates N task instances
- ✓ Engine, tier and arguments per instance
- ✓ Engine config (shuffle partitions, cluster size)

✗ Not supported, and not needed: an upstream task inventing new operator types or edges mid-run.

Stable logical DAG, adaptive physical execution — exactly how a database separates the query from its plan.

Borrowed from query optimizers and learned schedulers

Mid-query re-optimization
Kabra and DeWitt, SIGMOD 1998. Re-plan when observed statistics contradict estimates.

Eddies
Avnur and Hellerstein, SIGMOD 2000. Route work adaptively instead of fixing one order.

Progressive optimization
Markl et al., SIGMOD 2004. Checkpoints that validate cardinality and trigger re-planning.

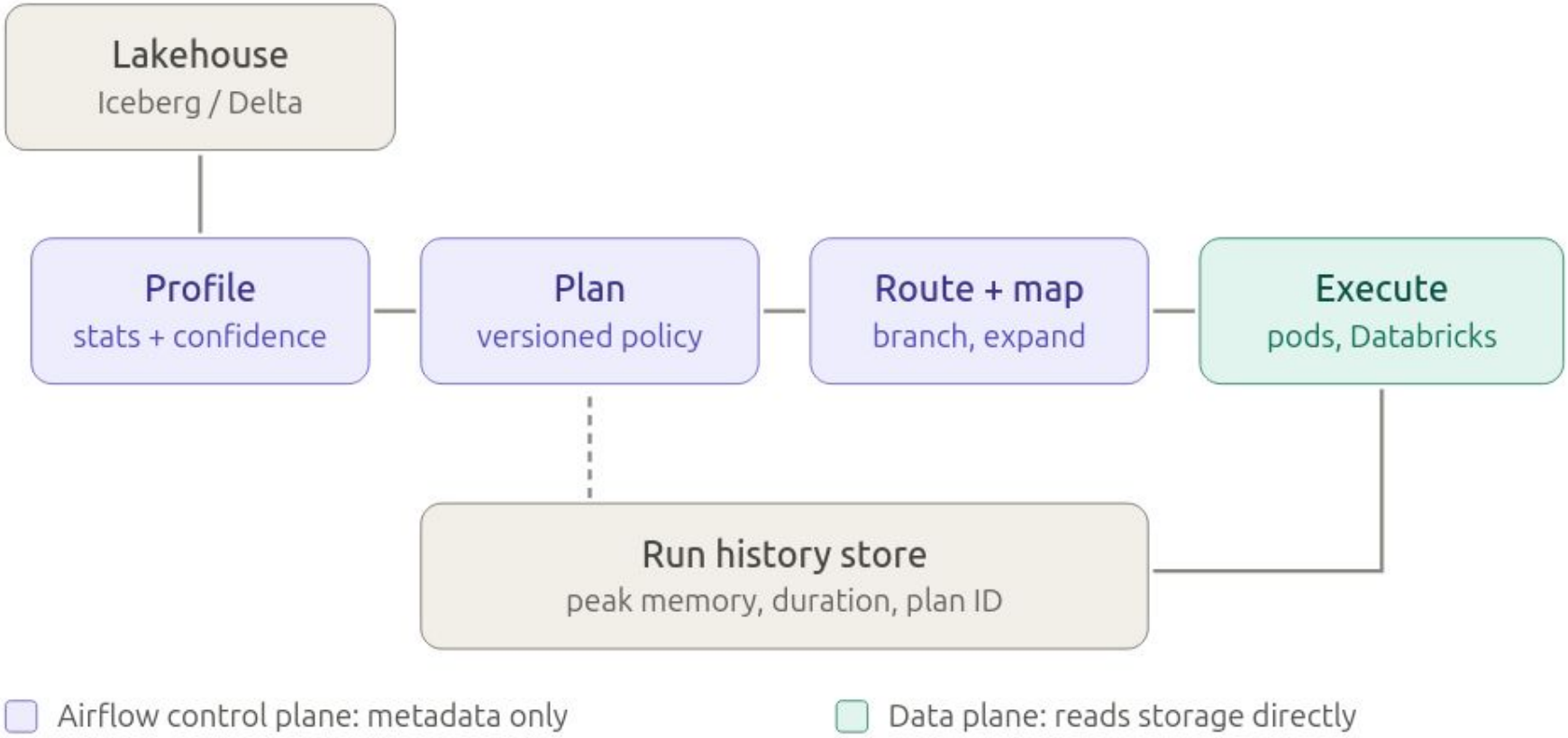
Spark AQE
Spark 3.0. Re-plans at shuffle boundaries: coalesce, switch joins, split skew.

Decima
Mao et al., SIGCOMM 2019. Learned scheduling and parallelism for data-processing DAGs.

Bao
Marcus et al., SIGMOD 2021. Learns optimizer hints from observed runtimes.

The loop: observe → estimate → select → execute → learn. Airflow task boundaries become our re-optimization checkpoints.

Reference architecture



One run: pin, profile, plan, expand, execute, commit

1 · Pin
Input snapshot ID

2 · Profile
Bytes, files, skew, confidence

3 · Plan
Policy version, engine, tier, N

4 · Expand
Bounded work units

5 · Execute
Pods or Databricks

6 · Commit
Validate, then publish once

> **Invariant**
> The plan is immutable for the DagRun. Retries and reruns read the same snapshot and the same plan. Resource escalation on OOM is part of the plan, not a new decision.

What the planner needs to know

Volume
Rows and compressed bytes for exactly this run's slice.

Partition skew
Largest ÷ median partition. Predicts stragglers and single-task OOM.

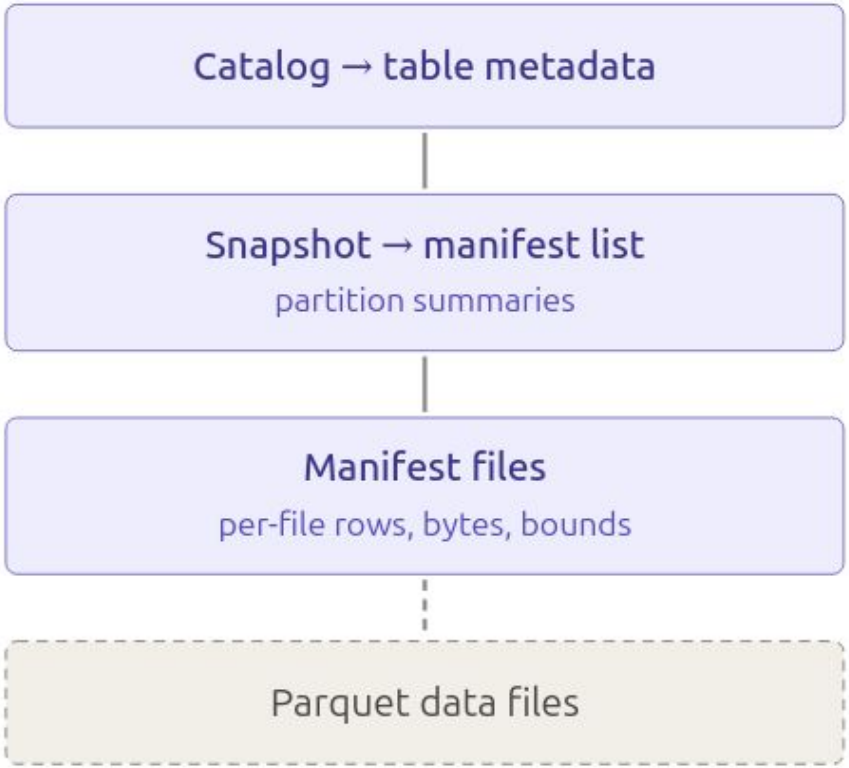
Cardinality
Distinct join or group keys. Drives broadcast vs shuffle and hash memory.

File layout
File count and small-file ratio. Drives split planning.

Quality gate: no blind trust
Track freshness and coverage of stats as a confidence level. Missing or stale stats → sample safely, or route to a conservative tier.

Budget: profiling costs seconds and never scans the data it's about to process.

Read the manifest, not the data



Profiler reads here

Kilobytes to megabytes

Delta equivalent: _delta_log add actions with numRecords

Never touched

Gigabytes to terabytes

Optional: Iceberg Puffin files can carry theta sketches for NDV estimates

Code: profile before heavyweight work

```
from collections import defaultdict
from statistics import median
from airflow.sdk import task

@task
def profile(table: str, ds: str, run_id: str) -> dict:
    from pyiceberg.catalog import load_catalog
    tbl = load_catalog("prod").load_table(table)
    snap = tbl.current_snapshot()
    scan = tbl.scan(row_filter=f"event_date = '{ds}'",
                    snapshot_id=snap.snapshot_id)
    files, parts, deletes = [], defaultdict(int), False
    for t in scan.plan_files():          # manifests only
        f = t.file
        files.append({"path": f.file_path, "part": str(f.partition),
                      "rows": f.record_count, "bytes": f.file_size_in_bytes})
        parts[str(f.partition)] += f.file_size_in_bytes
        deletes |= bool(t.delete_files)
    sizes = sorted(parts.values()) or [0]
    return {
        "snapshot_id": snap.snapshot_id,          # pin downstream
        "rows": sum(f["rows"] for f in files),    # upper bound if deletes
        "bytes": sum(sizes),
        "skew": sizes[-1] / max(median(sizes), 1),
        "files_uri": write_manifest(files, run_id),   # per-file list
        "confidence": stats_confidence(snap, deletes),
    }
```

Helpers like `write_manifest` and `stats_confidence` are yours to implement; the Airflow and PyIceberg calls are real.

The plan is a versioned contract

```
{
  "plan_id": "orders_daily/2026-11-27/p7",
  "policy_version": "p7",
  "snapshot_id": 7842119034,
  "files_uri": "s3://plans/orders_daily/2026-11-27/files.json",
  "engine": "pods",
  "target_bytes": 805306368,
  "units": 42,
  "max_units": 64,
  "tier": "M",
  "escalation": ["M", "L"],
  "max_concurrency": 32,
  "shuffle_parts": null,
  "confidence": "high",
  "reasons": ["31 GiB > local max 20 GiB",
              "skew 9.2: hot partition split into 3 units"]
}
```

Reproducible

Same snapshot + same policy ⇒ same choice.

Bounded

Units, concurrency, tier and escalation are capped.

Auditable

Persist what was chosen and why.

Rewrite the query, not just the resources

Small dimension

dim_product under 256 MiB → broadcast join. The fact side never shuffles.

Many rows per key

Low key cardinality → aggregate before the join. The shuffle shrinks.

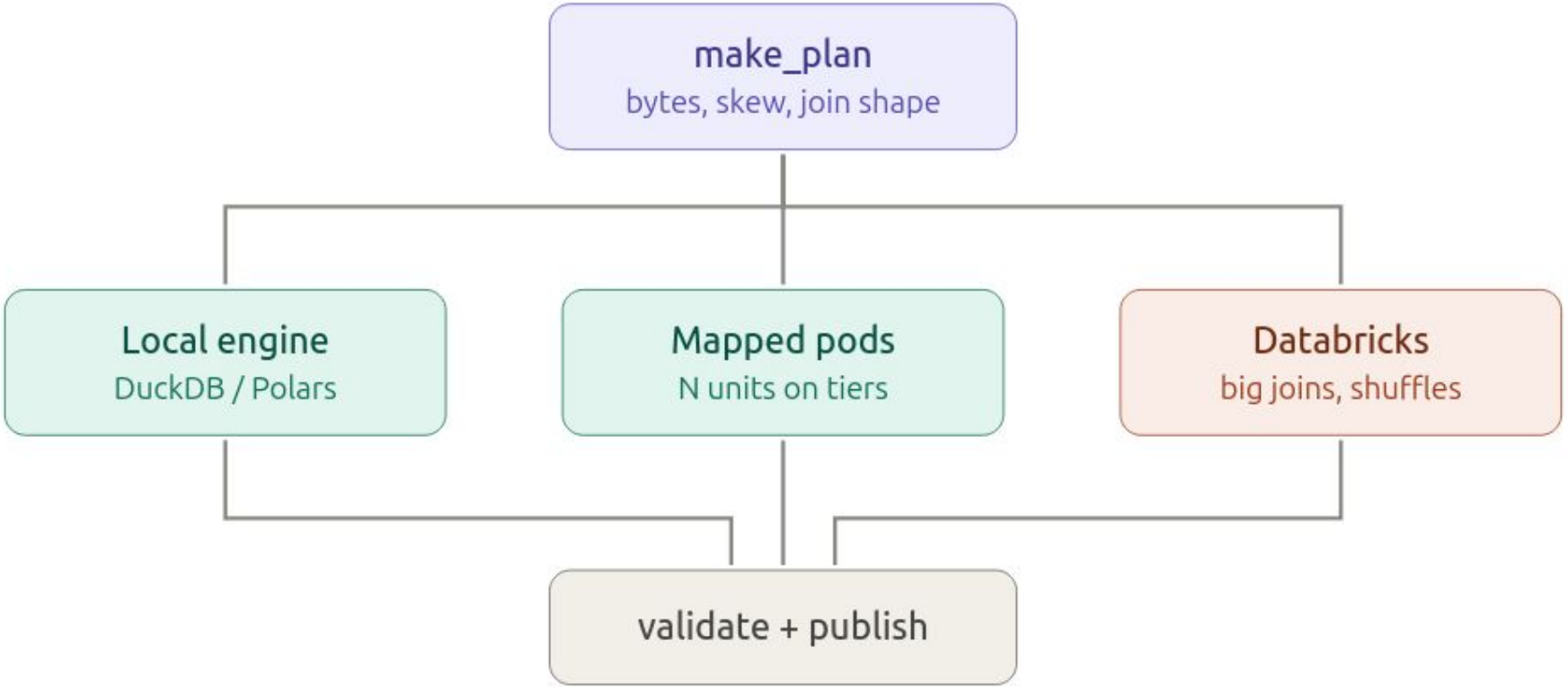
Few partitions changed

Under 10% touched → MERGE only those, not a full overwrite.

```
-- orders_daily.sql.j2: one logical query, several physical forms
WITH f AS (
  SELECT sku_id, {{ 'sum(qty) AS qty' if p.preagg else 'qty' }}
  FROM orders VERSION AS OF {{ p.version }}
  WHERE event_date = '{{ ds }}'
  {{ 'GROUP BY sku_id' if p.preagg else '' }}
)
SELECT {{ '/*+ BROADCAST(d) */' if p.broadcast else '' }}
       d.brand, sum(f.qty) AS units
FROM f JOIN dim_product d USING (sku_id)
GROUP BY d.brand
```

Same result, different physical plan. Each rewrite fired is recorded in the plan's reasons.

Pick the engine after looking



Route on workload shape — memory, operator complexity, startup latency, SLA — not row count alone. Same output schema and checks on every branch.

Code: a cost-based policy chooses the branch

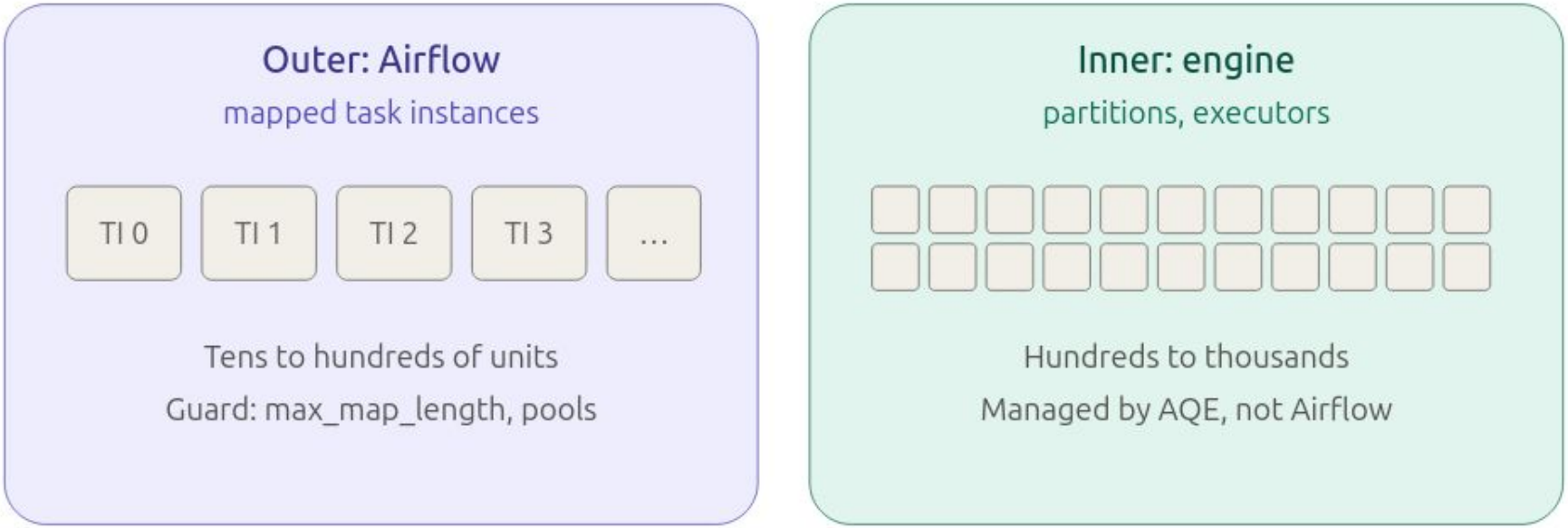
```
@task
def make_plan(stats: dict, plan_key: str, policy: str) -> dict:
    if prior := load_plan(plan_key):     # rerun or clear: same plan
        return prior
    pol = load_policy(policy)            # version pinned per DagRun
    if stats["confidence"] == "low":
        return save_plan(pol.conservative(stats, plan_key))
    ok = [e for e in pol.engines
          if e.peak_mem(stats) <= e.max_mem
          and e.runtime(stats) <= pol.sla_s]
    if not ok:                           # nothing fits: fail safe
        return save_plan(pol.conservative(stats, plan_key))
    best = min(ok, key=lambda e: e.cost(stats))
    return save_plan(best.plan(stats, plan_key, pol.version))

@task.branch
def choose_branch(plan: dict) -> str:
    return {"local": "run_local",
            "pods": "plan_units",
            "databricks": "databricks_spec"}[plan["engine"]]

@task(trigger_rule="none_failed_min_one_success")
def publish(plan: dict):
    validate_and_commit(plan["snapshot_id"], plan["plan_id"])
```

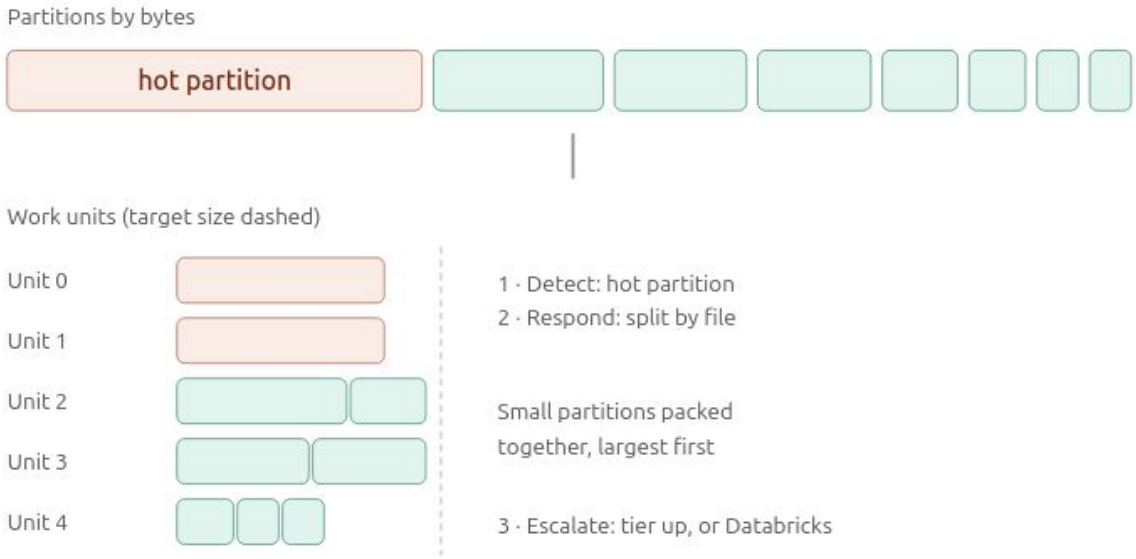
`load_policy`, `load_plan`, `save_plan` and the engine cost methods are your code, not library calls.

Two kinds of parallelism — don't confuse them



Anti-pattern: one Airflow task per shuffle partition

Bin-pack partitions into right-sized units




```
# inside each unit: files set the work boundary, Iceberg sets the rows
from pyiceberg.io.pyarrow import ArrowScan
scan = tbl.scan(row_filter=f"event_date = '{ds}'", snapshot_id=snapshot_id)
mine = [t for t in scan.plan_files() if t.file.file_path in unit_files]
rows = ArrowScan(tbl.metadata, tbl.io, scan.projection(),
                 scan.row_filter).to_table(mine)   # applies deletes
```

Never read a unit's Parquet files raw: deletes live in separate files. Check the `ArrowScan` signature against your PyIceberg version.

Code: planner emits units snapped to tiers

```
TIERS = {"S": 4096, "M": 16384, "L": 32768}   # MiB, predeclared

def snap_tier(pred_mb: float, headroom=1.2) -> str | None:
    for name, cap in TIERS.items():
        if pred_mb * headroom <= cap:
            return name
    return None                                  # too big for pods

@task
def plan_units(plan: dict) -> list[dict]:
    model = load_memory_model(plan["policy_version"])
    files = read_manifest(plan["files_uri"])    # from profile
    units = split_hot(files, target=plan["target_bytes"])
    bins = first_fit_decreasing(units, plan["target_bytes"],
                                max_bins=plan["max_units"])
    out = []
    for i, b in enumerate(bins):
        tier = snap_tier(model.p95_mb(b.bytes))
        assert tier, "make_plan routes oversize units to Databricks"
        out.append({
            "arguments": ["--snapshot", str(plan["snapshot_id"]),
                          "--files", write_unit(plan["plan_id"], i, b.files),
                          "--plan-id", plan["plan_id"]],
            "tier": tier,
        })
    return out
```

`split_hot`, `first_fit_decreasing`, `read_manifest` and `write_unit` are your code.

Code: tiered pods with bounded escalation

```
from airflow.providers.cncf.kubernetes.operators.pod import (
    KubernetesPodOperator)
from kubernetes.client import V1ResourceRequirements as Res

SPEC = {"S": ("2", "4Gi"), "M": ("4", "16Gi"), "L": ("8", "32Gi")}
ORDER = ["S", "M", "L"]

class TieredPodOperator(KubernetesPodOperator):
    def __init__(self, *, tier: str, **kw):
        super().__init__(**kw)
        self.tier = tier

    def execute(self, context):
        # plan is fixed; one step up on retry is part of the plan
        # production: step up only if the last attempt was OOMKilled
        step = min(context["ti"].try_number - 1, 1)
        t = ORDER[min(ORDER.index(self.tier) + step, 2)]
        cpu, mem = SPEC[t]
        self.log.info("plan tier=%s running tier=%s", self.tier, t)
        self.container_resources = Res(
            requests={"cpu": cpu, "memory": mem},
            limits={"memory": mem})
        return super().execute(context)

transform = TieredPodOperator.partial(
    task_id="transform", image=IMAGE, retries=1,
    pool="adaptive_compute", max_active_tis_per_dag=32,
).expand_kwargs(plan_units(plan))
```

The Databricks path: size the cluster, not Airflow

Estimated shuffle	Target per partition	Initial partitions	Cluster tier
120 GiB	512 MiB	240	L · 16

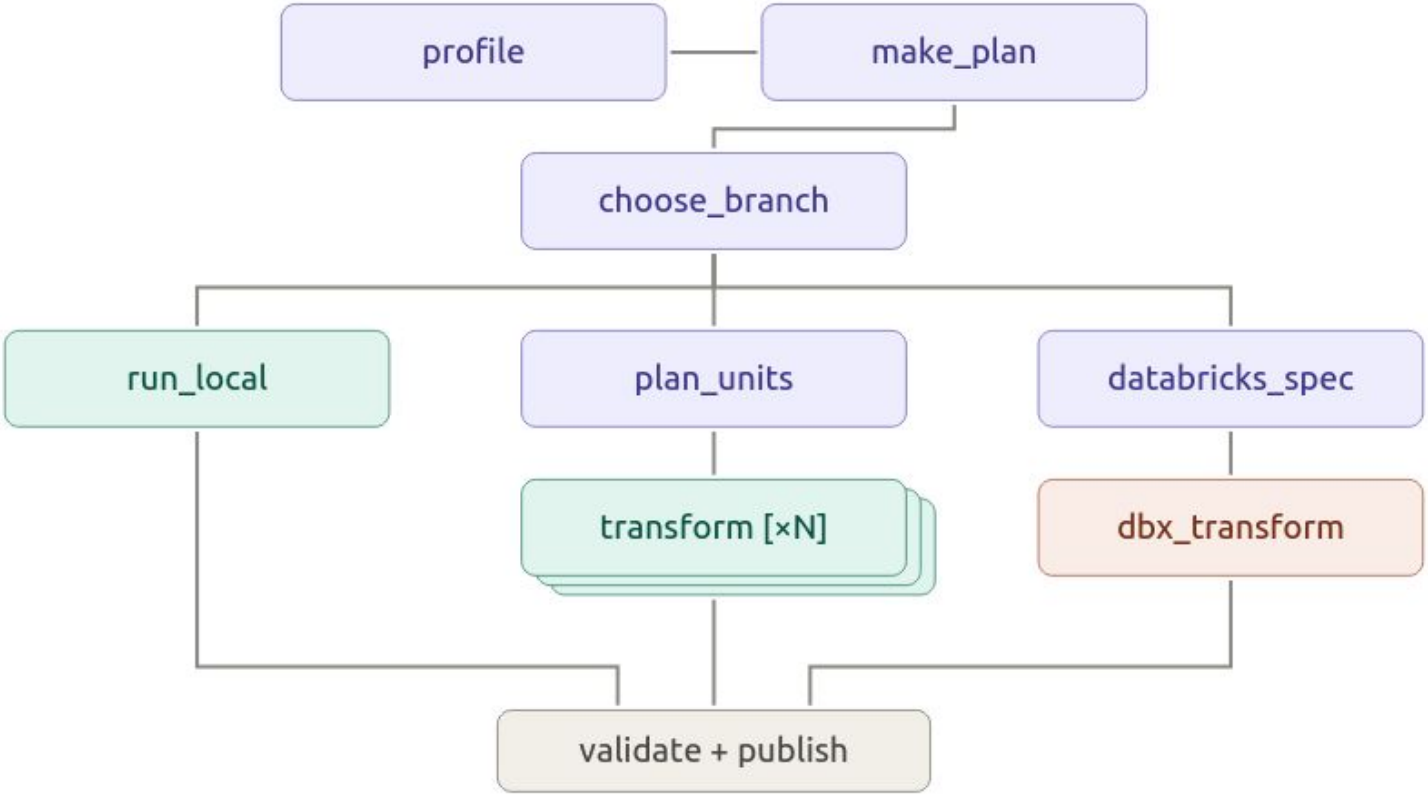
```
from airflow.providers.databricks.operators.databricks import (
    DatabricksSubmitRunOperator)

CLUSTERS = {"M": {"node_type_id": "r6id.xlarge",  "num_workers": 4},
            "L": {"node_type_id": "r6id.2xlarge", "num_workers": 16}}

@task
def databricks_spec(plan: dict) -> list[dict]:
    return [{"run_name": plan["plan_id"], "tasks": [{
        "task_key": "transform",
        "new_cluster": {
            **CLUSTERS[plan["tier"]],          # predeclared tiers
            "spark_version": SPARK_VERSION,
            "spark_conf": {
                "spark.sql.shuffle.partitions": str(plan["shuffle_parts"]),
                "spark.sql.adaptive.enabled": "true",
                "spark.sql.adaptive.skewJoin.enabled": "true"}},
        "notebook_task": {"notebook_path": NOTEBOOK,
            "base_parameters": {"snapshot_id": str(plan["snapshot_id"]),
                                "plan_id": plan["plan_id"]}}}]}]

dbx_transform = DatabricksSubmitRunOperator.partial(
    task_id="dbx_transform", databricks_conn_id="databricks",
    deferrable=True,
).expand(json=databricks_spec(plan))
```

The full adaptive DAG



Same DAG file every day; the branch taken and N change per run

Close the loop: from heuristics to a learned sizer

```
# inside every work unit: local or pod (Databricks: see note)
def emit_metrics(plan_id: str, unit: int, in_bytes: int, t0):
    rss_mb = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss / 1024
    write_history({"plan_id": plan_id, "unit": unit,
                   "engine": ENGINE, "input_bytes": in_bytes,
                   "peak_rss_mb": rss_mb,
                   "duration_s": time.monotonic() - t0})

# separate nightly DAG: refit, then ship as a new policy version
@task
def refit(current: str) -> str:
    h = read_history(days=30)
    coef = fit_quantile(h, y="peak_rss_mb", x="input_bytes",
                        by=["table", "engine"], q=0.95)
    return publish_policy(next_version(current), coef,
                          mode="shadow")
```

`write_history`, `fit_quantile` and `publish_policy` are your code.

What the planner decides

Rows 50,119

Skew 2×

Input size	Engine	Airflow units	Tier
6 MiB	Local	1	S

DuckDB or Polars in one pod; starts in seconds, no cluster. One process reads everything, so partition skew doesn't matter here.

Illustrative assumptions: 120 bytes per row, 768 MiB target per unit, 3× memory amplification plus 512 MiB overhead, 20% headroom, tiers S 4 GiB · M 16 GiB · L 32 GiB, local engine limited to S and M. 24 partitions, largest = skew × median; partitions over 2× target are split by file.

Make it safe before making it clever

Pin
Read a specific snapshot; reuse the same plan on retry and rerun.

Cap
Bound map length, active instances, pool slots, tier, runtime and spend.

Fall back
Missing or stale stats → conservative tier or targeted sampling.

Commit once
Idempotent units write to staging; one validated publish.

Observe
Record engine, tier, confidence, peak RAM, cost and SLA per plan ID.

Kill switch
One flag forces the static plan for any DAG.

Stay cheap
Fold profile and plan into one task; run the local branch on a warm worker. Track the added latency.

XCom carries plans and metrics only. Branch joins need deliberate trigger rules.

Roll out by evidence, not overnight

1 · Observe
Collect volume, skew, peak RSS, failures

2 · Shadow
Log the proposed plan; keep static execution

3 · Canary
Enable on a few low-risk DAGs

4 · Expand
Retune from telemetry; widen scope

Scorecard against the static baseline

P95 run latency · cost per successful GiB · OOM rate · retries · planning overhead · scheduler pressure

Report measured results only — never hypothetical savings.

Three things to take home

Profile
Use pinned table metadata first; sample only when you must.

Route
Keep the DAG stable; let a versioned plan choose the query form and the physical execution.

Resize
Size Airflow units to tiers and engine partitions to bytes — and learn from every run.

Start small: add one profile task and one branch, in shadow mode, to your most volatile DAG next sprint.

Appendix: references

Airflow

Dynamic task mapping · Branching and trigger rules · XComs · KubernetesPodOperator and Databricks provider docs (airflow.apache.org)

Table formats

Iceberg spec: manifests, delete files and Puffin files (iceberg.apache.org) · Delta Lake transaction log protocol and data skipping (delta.io)

Engines

Spark SQL performance tuning: adaptive query execution · Databricks job clusters and autoscaling · DuckDB memory and spilling · Polars streaming engine

Papers

Kabra and DeWitt, SIGMOD 1998 · Avnur and Hellerstein, SIGMOD 2000 · Markl et al., SIGMOD 2004 · Mao et al. (Decima), SIGCOMM 2019 · Marcus et al. (Bao), SIGMOD 2021

Implementation details are version-sensitive; verify against your pinned Airflow and provider releases.

Appendix: Delta variant and cardinality

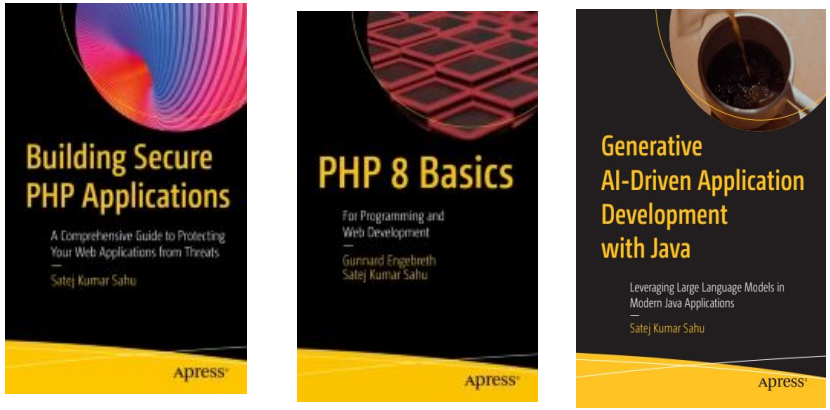
```
import duckdb
from deltalake import DeltaTable

def profile_delta(uri: str, ds: str) -> dict:
    dt = DeltaTable(uri)
    df = dt.get_add_actions(flatten=True).to_pandas()
    df = df[df["partition.event_date"] == ds]
    by_part = df.groupby("partition.region")["size_bytes"].sum()
    return {"version": dt.version(),        # pin downstream
            "rows": int(df["num_records"].sum()),
            "bytes": int(df["size_bytes"].sum()),
            "skew": by_part.max() / max(by_part.median(), 1)}

def cardinality_class(files: list[str], key: str) -> str:
    # no sketches available: classify from a sample, don't estimate NDV
    n, d = duckdb.sql(f"""
        SELECT count(*), approx_count_distinct({key})
        FROM read_parquet({files[:50]!r})
        USING SAMPLE 1 PERCENT (bernoulli)""").fetchone()
    return "high" if d > 0.5 * n else "low"
```

About Me

- *Solution Architect* at Zalando SE with over 16 years of experience in the industry.
- Previously worked at organizations such as *Boeing, Adidas, Honeywell* specializing in architecture, big data and machine learning use cases.
- Author : "*Building Secure PHP Applications*" and "*PHP 8 Basics: For Programming and Web Development*" with *Apress / Springer* publications and "*Generative AI driven Application Development with Java*" Book



Books from the Speaker

Blog Github LinkedIn Youtube


A
Q