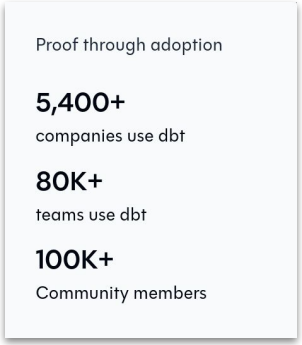
Optimizing SQL Data Pipelines

Drew Ripberger

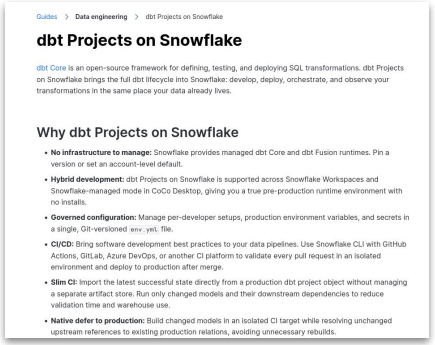
aripberg@cs.cmu.edu

SQL Pipelines

- Today we are talking about external orchestration systems that run SQL queries
- There are many vendors offering these frameworks…
  - dbt
  - SQLMesh
  - Google's DataForm
  - Databricks' LakeFlow
- The "T" in ETL
- They support a variety of DBs



https://www.getdbt.com/



https://docs.snowflake.com/en/user-guide/data-engineering/dbt-projects-on-snowflake

The Transformation Graph

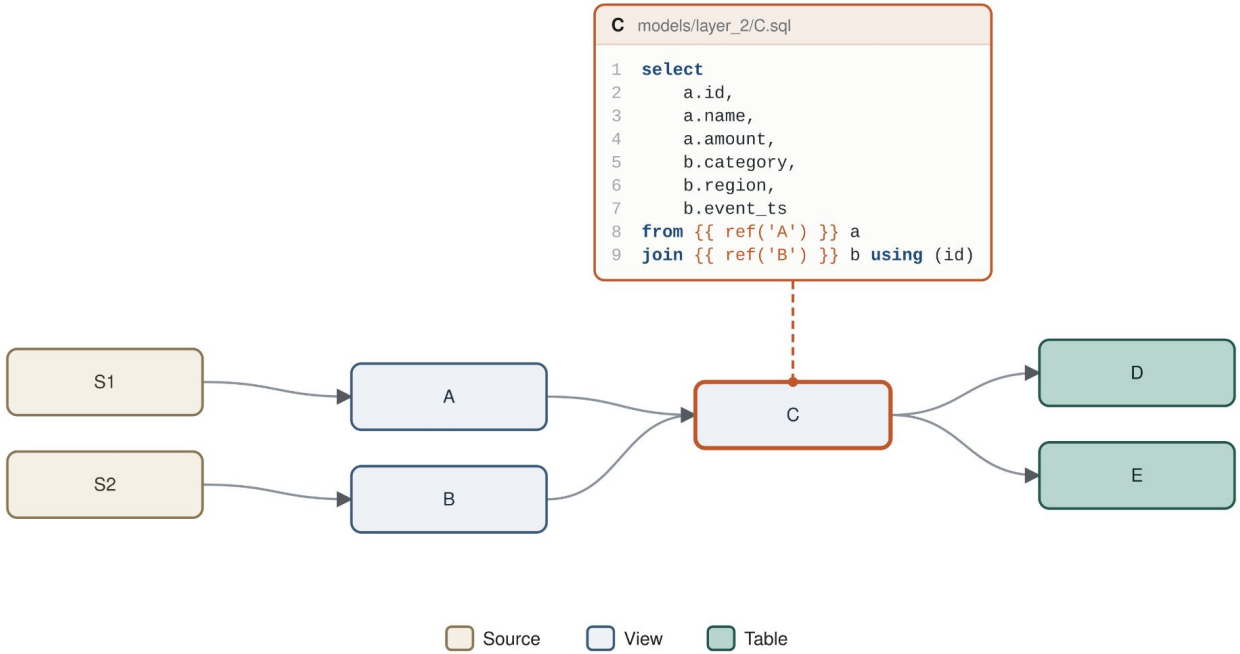
```
CREATE VIEW A;
```

```
CREATE VIEW B;
```

```
CREATE VIEW C;
```

```
CREATE TABLE D;
```

```
CREATE TABLE E;
```



The Transformation Graph

```
CREATE VIEW A;
```

```
CREATE VIEW B;
```

```
CREATE VIEW C;
```

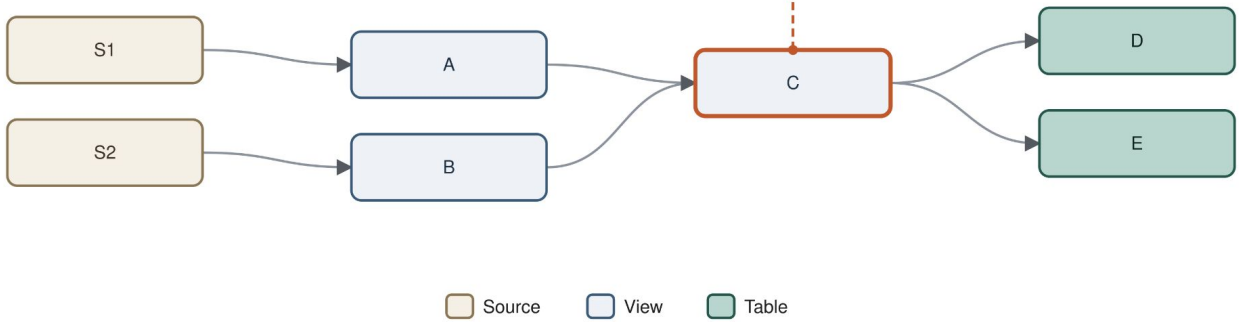
```
CREATE TABLE D;
```

```
CREATE TABLE E;
```

How many times does the JOIN between A and B need to be computed?

C models/layer_2/C.sql

```sql
select
    a.id,
    a.name,
    a.amount,
    b.category,
    b.region,
    b.event_ts
from {{ ref('A') }} a
join {{ ref('B') }} b using (id)
```



Our Optimizer

- Adaptive Materialization
- Pushdown
- Degree of Parallelism Tuning
- Node Fusion
- Logical Rewriting

Our Optimizer

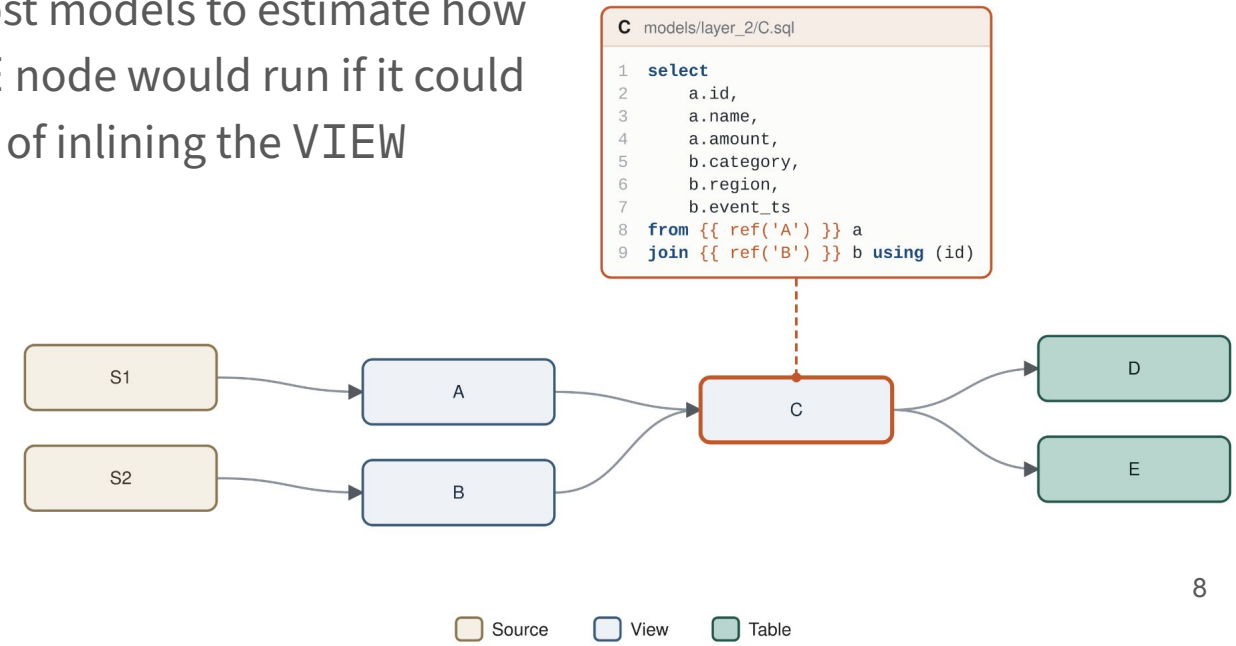
- Adaptive Materialization ⟵ Let’s start here
- Pushdown
- Degree of Parallelism Tuning
- Node Fusion
- Logical Rewriting

Our Optimizer - Adaptive Materialization

- **Goal:** Locate where `VIEW` usage introduces duplicate computation
- **Problem:** It's not always helpful to materialize relations just to reduce redundancy
  - Sometimes the `VIEW` has an extremely large output
  - The `VIEW` can optimize extremely well once inlined
  - Materializing a common node requires downstream nodes to block waiting on the query
- **Partial Solution:** Continuously collect and learn information about how each query in our DAG performs

Our Optimizer - Adaptive Materialization

- EXPLAIN can give us details about plans
- Piggyback on DAG runs to learn characteristics of them
- We try to build simple cost models to estimate how much faster each TABLE node would run if it could use a Table Scan instead of inlining the VIEW



Our Optimizer - Adaptive Materialization

Every member v of S goes into the query as its own MATERIALIZED cte(v),
so EXPLAIN names the plan region that is v and the cost model can price it.

1 What the consumers pay for S today

$$\text{pay}(c) = \sum_{v \text{ seen by } c} \text{cost}(\ v \text{ inside } c\ )$$

$$\text{total} = \sum_{c} \text{pay}(c)$$

2 What building S once would cost instead

$$\text{build\_once} = \sum_{v \text{ in } S} \text{cost}(\ v \text{ on its own}\ )$$

$$\text{write}(S) = \text{what writing those rows out costs}$$

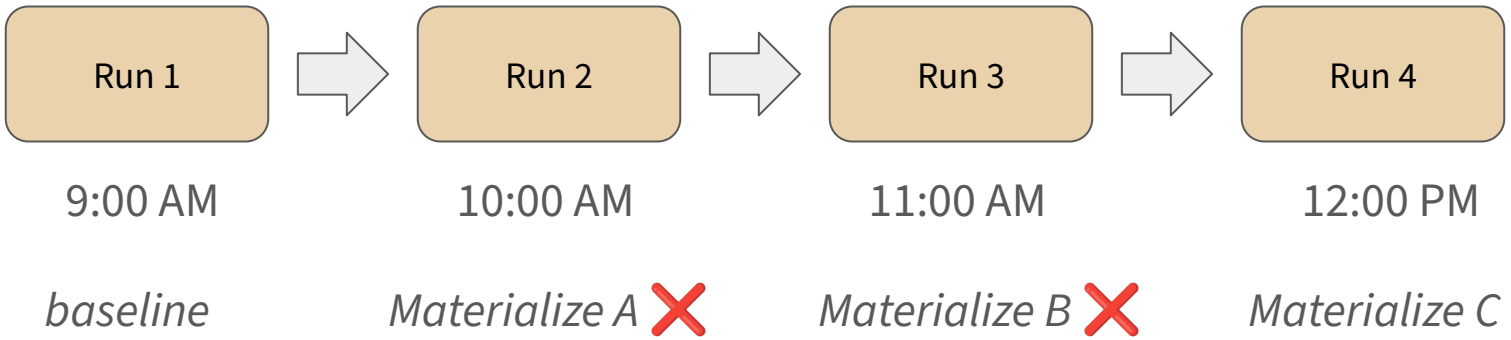
3 What materializing S would change

$$\textbf{duplicate} = \text{total} - \text{build\_once}$$

$$\textbf{net gain} = \text{duplicate} - \text{write}(S)$$

Every consumer computes S today; materializing it keeps exactly one of those builds.

Our Optimizer - Adaptive Materialization



dag-bench

- There is no good public academic or industry benchmark for “DAGs”
- TPC-DI…
- We put together a benchmark of 10 dbt projects
- Focuses on varying topology and SQL (not necessarily size)
- Each have a corresponding “real world” application
- Synthetic data generator

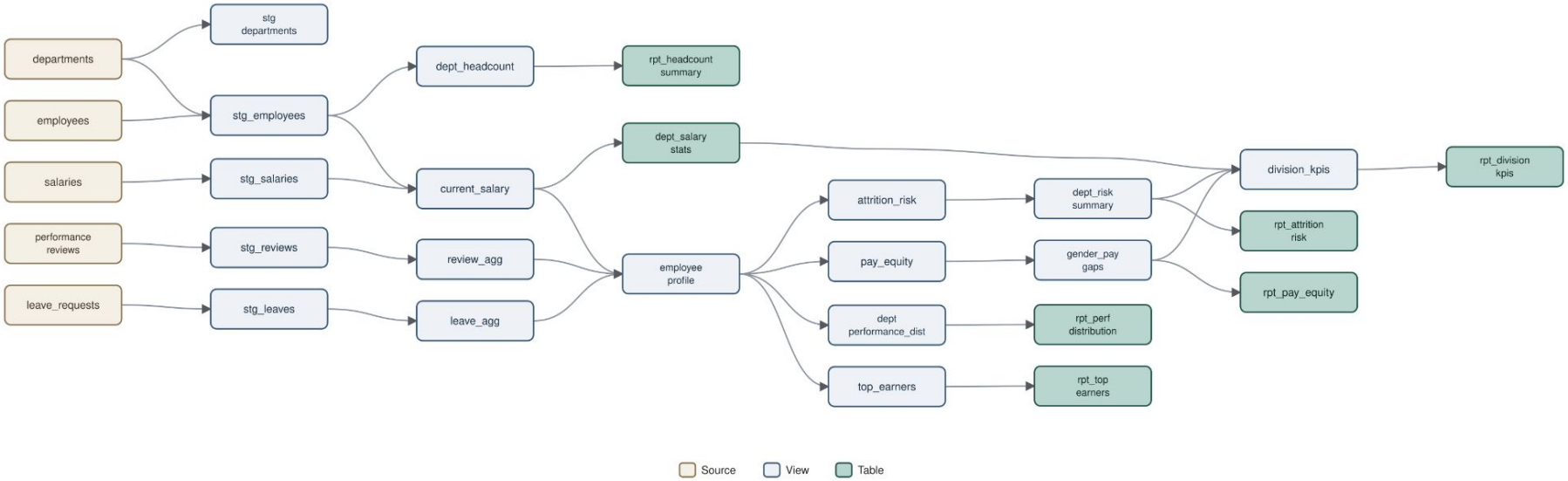
https://github.com/drewrip/dag-bench

dag-bench

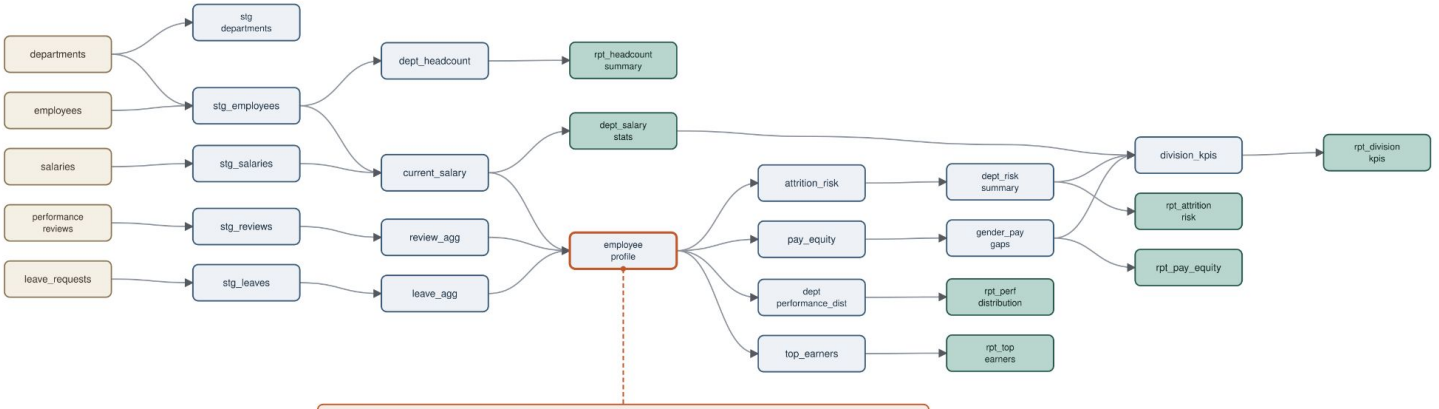
Project	Domain	Sources	Nodes	Views	Tables	Depth	Branching Nodes
p01_iot	IoT fleet monitoring	4	12	11	1	5	2
p02_adtech	Digital advertising funnel	4	10	9	1	6	3
p03_ecommerce	Online retail	7	25	20	5	7	6
p04_fraud	Card/banking fraud detection	4	22	16	6	6	9
p05_hr	People analytics	5	24	17	7	7	5
p06_logistics	Supply chain / warehouse logistics	5	23	18	5	8	6
p07_saas	B2B SaaS product analytics	5	17	13	4	6	4
p08_healthcare	Health insurance claims	5	20	14	6	6	5
p09_gaming	Mobile/video game analytics	5	20	16	4	6	3
p10_energy	Electric utility / smart grid	4	23	16	7	7	11

dag-bench

p05 - HR



dag-bench



p05 - HR

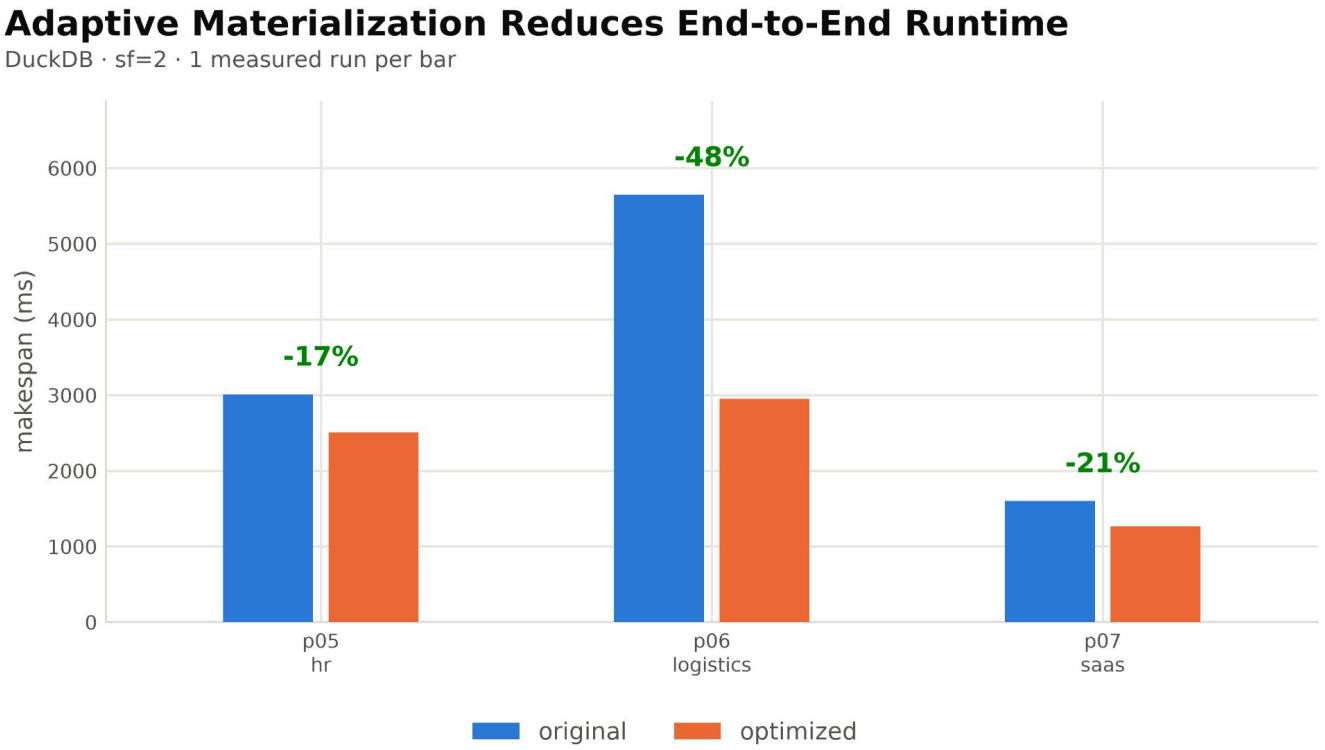
employee_profile models/layer_3/employee_profile.sql

```sql
with current_salary as (

    select * from {{ ref('current_salary') }}

),

review_agg as (

    select * from {{ ref('review_agg') }}

),

leave_agg as (

    select * from {{ ref('leave_agg') }}

),

final as (

    select cs.emp_id, cs.full_name, cs.dept_name, cs.division, cs.location,
        cs.gender, cs.employment_type, cs.tenure_years, cs.is_active,
        cs.base_salary, cs.total_comp,
        coalesce(ra.avg_score,0) as avg_perf_score,
        coalesce(ra.review_count,0) as review_count,
        coalesce(la.total_leave_days,0) as total_leave_days,
        coalesce(la.sick_days,0) as sick_days,
        case when ra.avg_score>=4.0 then 'High' when ra.avg_score>=2.5 then 'Meets' else 'Below' en…
    from current_salary cs
    left join review_agg ra using (emp_id)
    left join leave_agg la using (emp_id)

)

select * from final
```

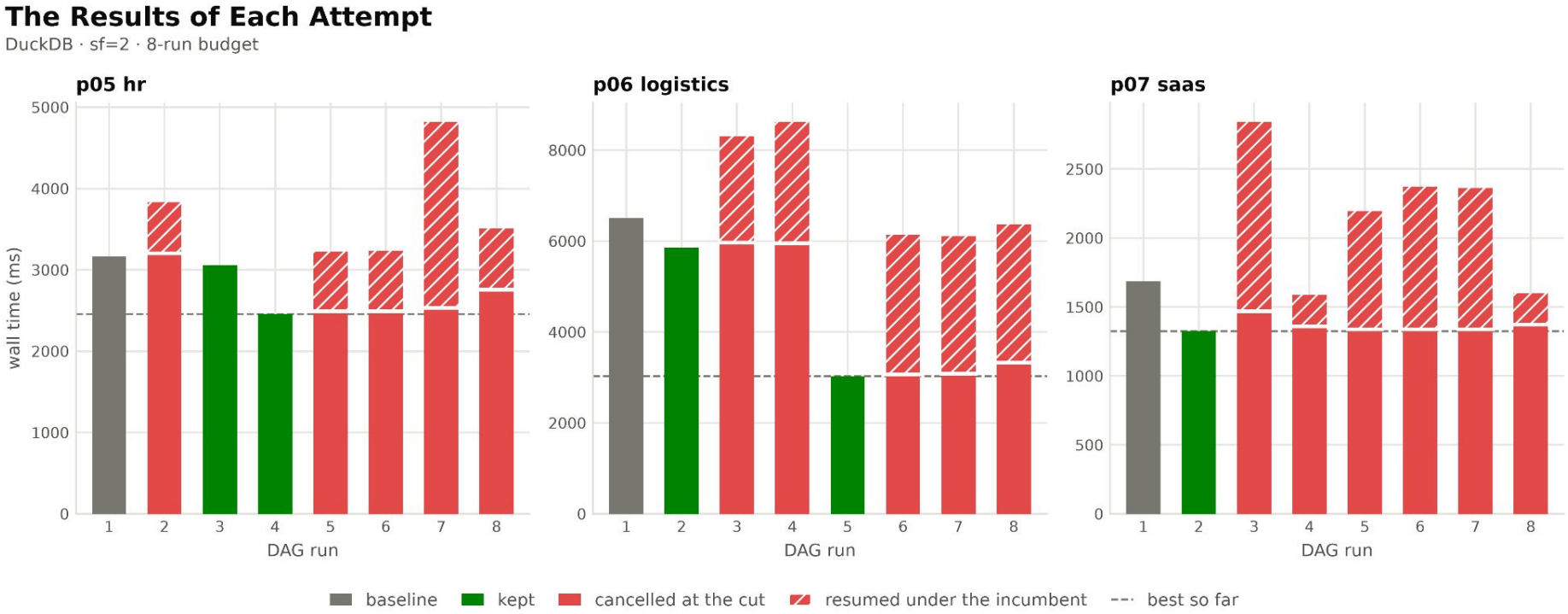
The Results

- Let’s look at a small scale example
- Running DAGs `p05`, `p06`, and `p07`
- `sf=2`, the source tables for each project are 2GB (compressed on disk)
- 8 threads, 16GB mem
- Each project is run 8 times
- What speedup do they achieve after these runs

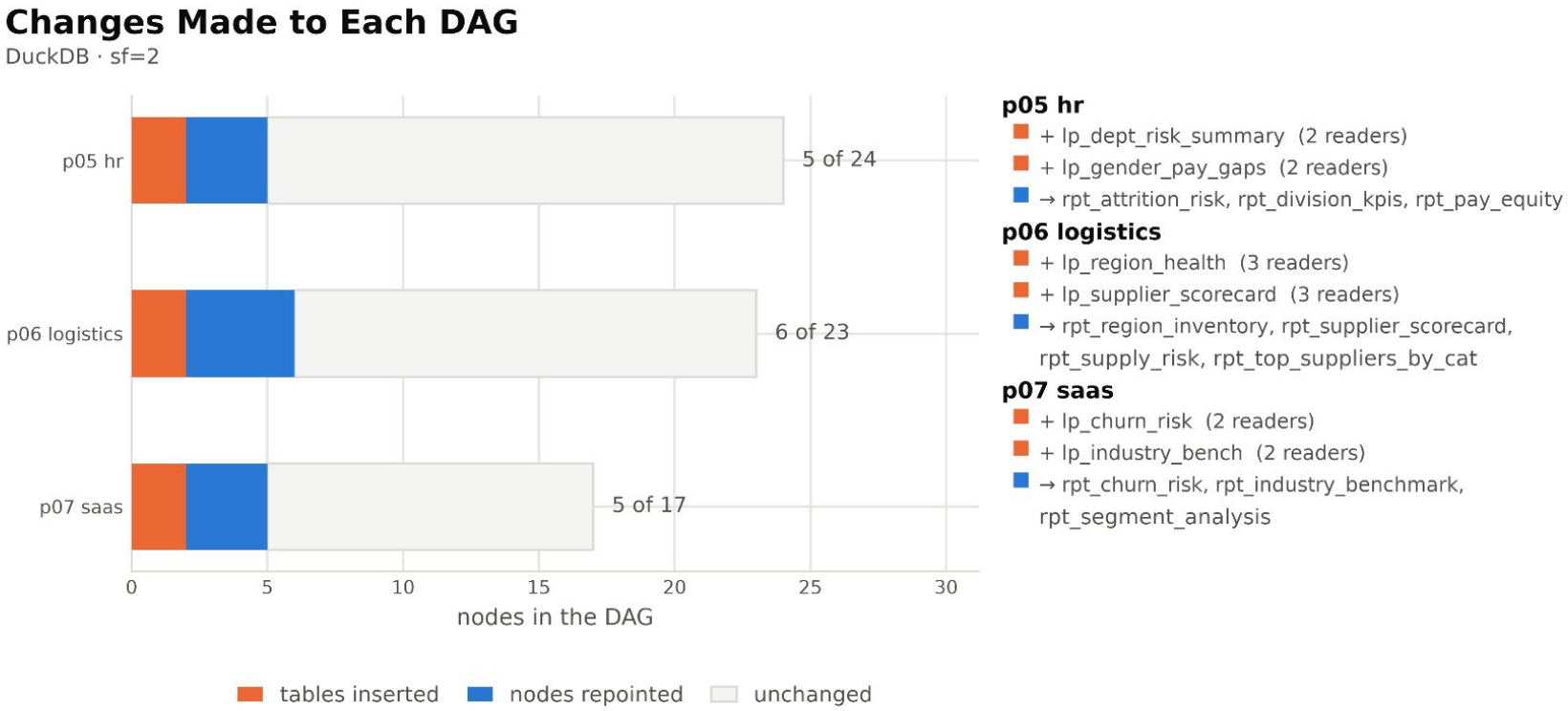
The Results


Adaptive Materialization Reduces End-to-End Runtime
DuckDB · sf=2 · 1 measured run per bar
makespan (ms)
0
1000
2000
3000
4000
5000
6000
-17%
-48%
-21%
p05
hr
p06
logistics
p07
saas
original
optimized

The Results


The Results of Each Attempt
DuckDB · sf=2 · 8-run budget
p05 hr
p06 logistics
p07 saas
wall time (ms)
DAG run
baseline
kept
cancelled at the cut
resumed under the incumbent
best so far

The Results


Changes Made to Each DAG
DuckDB · sf=2
p05 hr
p06 logistics
p07 saas
5 of 24
6 of 23
5 of 17
0
5
10
15
20
25
30
nodes in the DAG
tables inserted
nodes repointed
unchanged
p05 hr
+ lp_dept_risk_summary (2 readers)
+ lp_gender_pay_gaps (2 readers)
→ rpt_attrition_risk, rpt_division_kpis, rpt_pay_equity
p06 logistics
+ lp_region_health (3 readers)
+ lp_supplier_scorecard (3 readers)
→ rpt_region_inventory, rpt_supplier_scorecard, rpt_supply_risk, rpt_top_suppliers_by_cat
p07 saas
+ lp_churn_risk (2 readers)
+ lp_industry_bench (2 readers)
→ rpt_churn_risk, rpt_industry_benchmark, rpt_segment_analysis

Thanks!

aripberg@cs.cmu.edu

drewrip.github.io