ReSequel: Robust LLM-assisted Query Rewriting and Optimization using Templatization and Sampling

Saeed Fathollahzadeh [1], Essam Mansour [1], and Matthias Boehm [2,3]

1 Concordia University

2 TU Berlin

3 BIFOLD

Aug 31st – Sep 4th

VLDB 2026 BOSTON

Motivating Example — Declarative Query Processing

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

```
SELECT COUNT(*)
FROM PostHistory h,
     Posts p,
     Users u,
     Badges b
WHERE b.UserId = u.Id
     AND p.OwnerUserId = u.Id
     AND h.UserId = u.Id
     AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
     AND h.CreationDate <= '2014-09-10 08:22:43' ::timestamp
     AND p.PostTypeId = 2;
```

Motivating Example — Declarative Query Processing

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

```
1: SELECT COUNT(*)------  No early aggregation ⇒ huge intermediates
2: FROM PostHistory h,
3:      Posts p,
4:      Users u,
5:      Badges b
6: WHERE b.UserId = u.Id
7:      AND p.OwnerUserId = u.Id
8:      AND h.UserId = u.Id
9:      AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
10:     AND h.CreationDate <= '2014-09-10 08:22:43' ::timestamp
11:     AND p.PostTypeId = 2;
```

Motivating Example — Declarative Query Processing

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

```
1: SELECT COUNT(*)
2: FROM PostHistory h,
3:      Posts p,
4:      Users u,
5:      Badges b
6: WHERE b.UserId = u.Id
7:      AND p.OwnerUserId = u.Id
8:      AND h.UserId = u.Id
9:      AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
10:     AND h.CreationDate <= '2014-09-10 08:22:43' ::timestamp
11:     AND p.PostTypeId = 2;
```

No early aggregation ⇒ huge intermediates

Flat 4-way Cross Join

Motivating Example — Declarative Query Processing

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

```
1: SELECT COUNT(*)
2: FROM PostHistory h,
3:      Posts p,
4:      Users u,
5:      Badges b
6: WHERE b.UserId = u.Id
7:      AND p.OwnerUserId = u.Id
8:      AND h.UserId = u.Id
9:      AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
10:     AND h.CreationDate <= '2014-09-10 08:22:43' ::timestamp
11:     AND p.PostTypeId = 2;
```

No early aggregation ⇒ huge intermediates

Flat 4-way Cross Join

Join through **users** is redundant — it is only a PK bridge

Motivating Example — Declarative Query Processing

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

```
1:  SELECT COUNT(*)
2:  FROM PostHistory h,
3:       Posts p,
4:       Users u,
5:       Badges b
6:  WHERE b.UserId = u.Id
7:        AND p.OwnerUserId = u.Id
8:        AND h.UserId = u.Id
9:        AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
10:       AND h.CreationDate <= '2014-09-10 08:22:43' ::timestamp
11:       AND p.PostTypeId = 2;
```

No early aggregation ⇒ huge intermediates

Flat 4-way Cross Join

Join through **users** is redundant — it is only a PK bridge

Filters applied **after** the join

Motivating Example — Declarative Query Processing

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

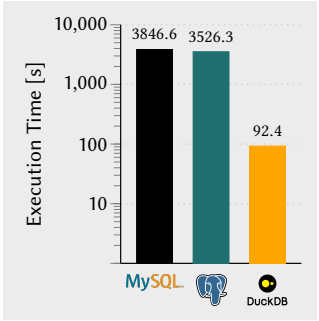
```
1: SELECT COUNT(*)
2: FROM PostHistory h,
3:      Posts p,
4:      Users u,
5:      Badges b
6: WHERE b.UserId = u.Id
7:      AND p.OwnerUserId = u.Id
8:      AND h.UserId = u.Id
9:      AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
10:     AND h.CreationDate <= '2014-09-10 08:22:43' ::timestamp
11:     AND p.PostTypeId = 2;
```

No early aggregation ⇒ huge intermediates

Flat 4-way Cross Join

Join through **users** is redundant — it is only a PK bridge

Filters applied **after** the join



Motivating Example — Declarative Query Processing

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

```
1: SELECT COUNT(*)
2: FROM PostHistory h,
3:      Posts p,
4:      Users u,
5:      Badges b
6: WHERE b.UserId = u.Id
7:       AND p.OwnerUserId = u.Id
8:       AND h.UserId = u.Id
9:       AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
10:      A
11:      A
```

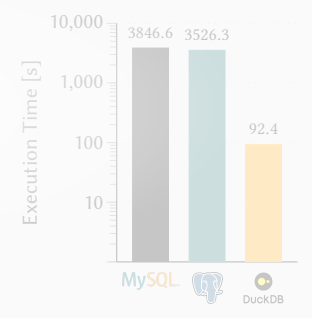
No early aggregation ⇒ huge intermediates

Flat 4-way Cross Join

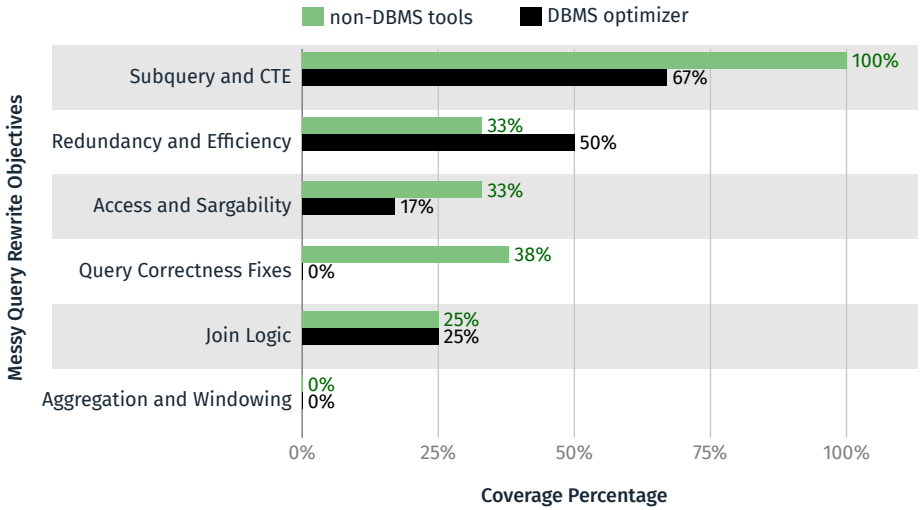
Join through **users** is redundant — it is only a PK bridge

The optimizer is not going to save you here

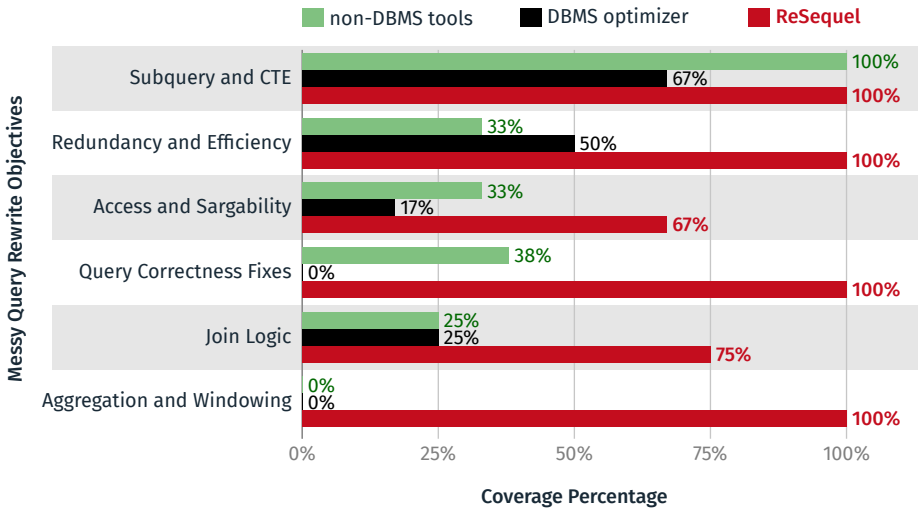
Filters applied **after** the join



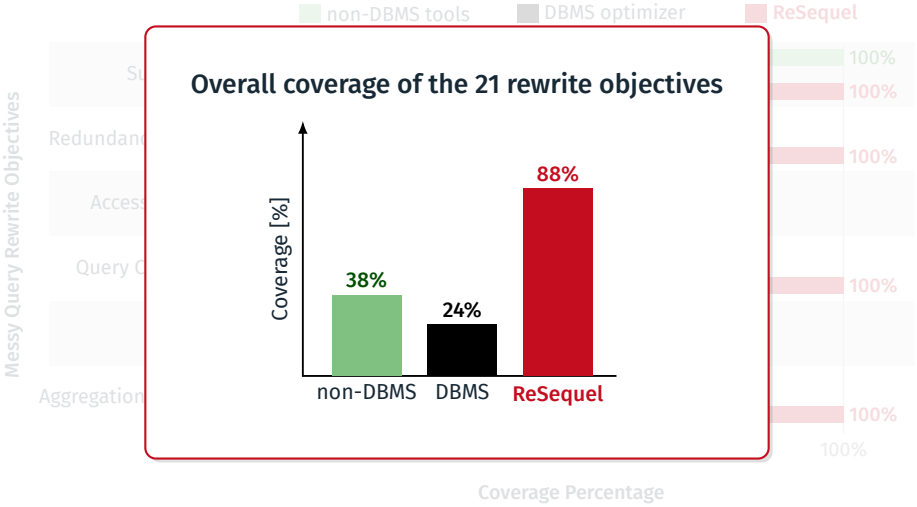
What Do Query Rewriters Actually Cover?



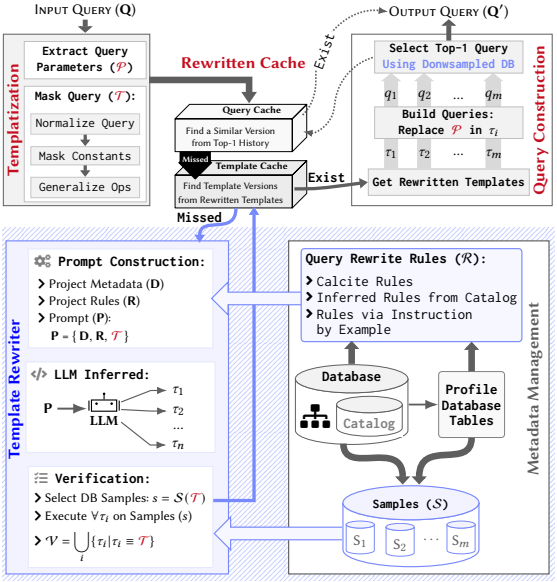
What Do Query Rewriters Actually Cover?



What Do Query Rewriters Actually Cover?



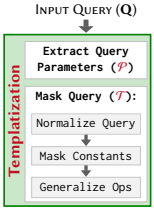
ReSequel Architecture



Key Design Decisions:

- Outer layer: any DBMS
- Template-based: rewrite once
- Metadata-guided: schema + stats
- Sample-verified: correctness check
- Cached: amortized over workload

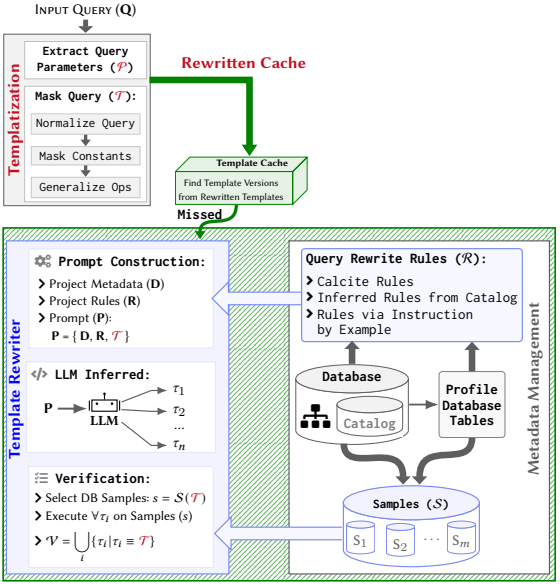
ReSequel Architecture



Templatization:

- **Normalize:** canonical form
- **Mask constants:** literals $\rightarrow$ placeholders
- **Generalize ops:** same shape, one key
- Output: template $\mathcal{T}$ + params $\mathbf{P}$
- Effect: many queries, one rewrite

ReSequel Architecture

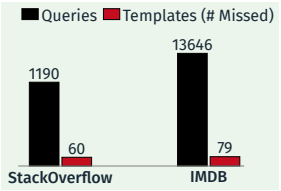


Metadata Management:

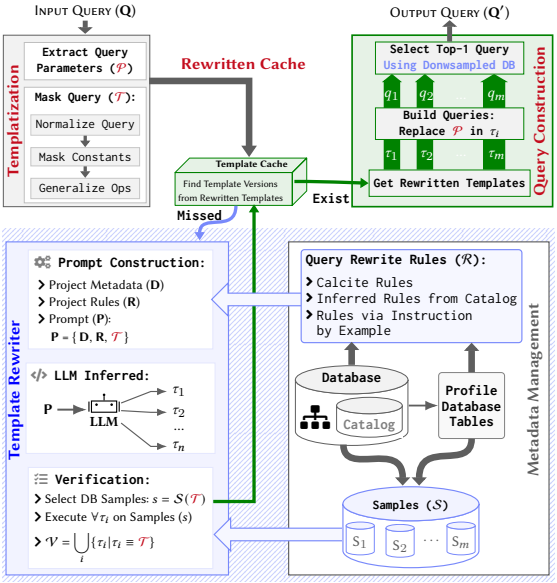
- **Catalog:** schema, keys, constraints
- **Profiling:** per-table/column statistics
- **Rules** $\mathcal{R}$: Calcite, catalog, by-example
- **Samples** $\mathcal{S}$: small, representative

Template Rewriter:

- **Prompt** $\mathbf{P}$: metadata + rules
- **LLM proposes** N candidate rewrites
- Verify on samples, not on full DB
- Output: verified set $\mathcal{V}$ **(equivalent only)**



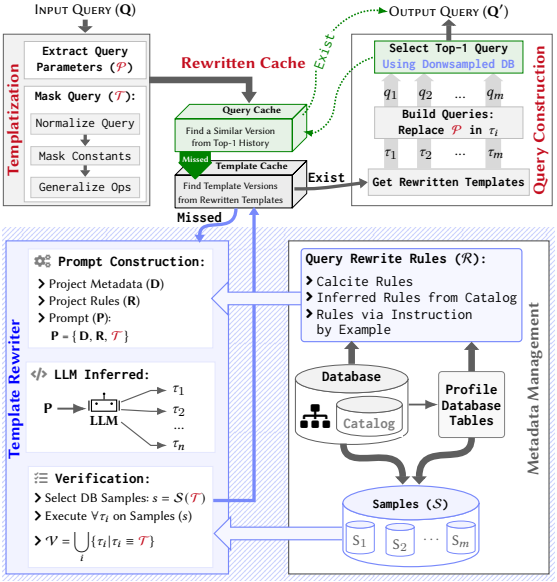
ReSequel Architecture



Query Construction:

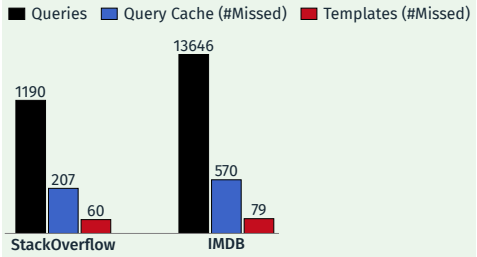
- Fetch verified templates $\tau_1, \ldots, \tau_m$
- **Bind params:** replace $\mathbf{P}$ in each τ_i
- **Rank candidates** on downsampled DB
- **Select Top-1** query $\mathbf{Q}'$
- DBMS executes $\mathbf{Q}'$ unchanged

ReSequel Architecture



Query Cache:

- **Lookup:** similar query in Top-1 history
- **Hit:** reuse the cached Top-1 rewrite
- **No LLM call**, no sample runs, no ranking
- Miss: fall back to the template cache
- Cost amortized over the workload



Recap — The Motivating Query

- SQL is **declarative**: the user states *what* to compute, the optimizer decides *how*.

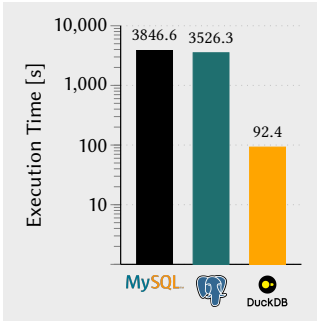
```
1: SELECT COUNT(*)
2: FROM PostHistory h,
3:      Posts p,
4:      Users u,
5:      Badges b
6: WHERE b.UserId = u.Id
7:       AND p.OwnerUserId = u.Id
8:       AND h.UserId = u.Id
9:       AND h.CreationDate >= '2010-07-27 18:08:19' ::timestamp
10:      AND h.CreationDate <= '2014-09-10 08:22:43' ::timestamp
11:      AND p.PostTypeId = 2;
```

No early aggregation ⇒ huge intermediates

Flat 4-way Cross Join

Join through **users** is redundant — it is only a PK bridge

Filters applied **after** the join



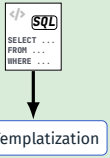
An Example Workflow

Qyery Template (Q122, Stats): ❶

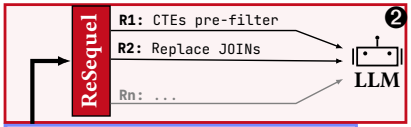
```
 1:SELECT COUNT(*)
 2:FROM History h,
 3:  Posts p,
 4:  Users u,
 5:  Badges b
 6:WHERE b.UserId = u.Id
 7:  AND p.OwnerUserId = u.Id
 8:  AND h.UserId = u.Id
 9:  AND h.CreationDate >= '$$$'
10:  AND h.CreationDate <= '$$$'
11:  AND p.PostTypeId = ###;
```

Masked Predicates

Rewriting pipeline:



An Example Workflow



Qyery Template (Q122, Stats): ❶

```
 1:SELECT COUNT(*)
 2:FROM History h,
 3:  Posts p,
 4:  Users u,
 5:  Badges b
 6:WHERE b.UserId = u.Id
 7:  AND p.OwnerUserId = u.Id
 8:  AND h.UserId = u.Id
 9:  AND h.CreationDate >= '$$$'
10:  AND h.CreationDate <= '$$$'
11:  AND p.PostTypeId = ###;
```

Masked Predicates

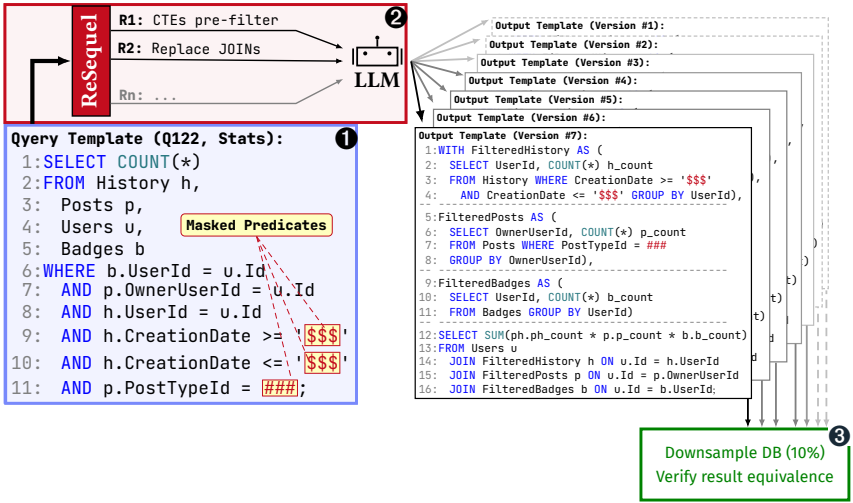
Rewriting pipeline:

❶ Templatization

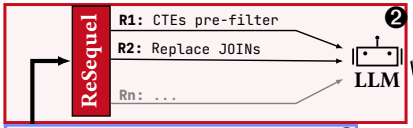
❷ Template Rewriter

Prompt LLM → *N* variants

An Example Workflow



An Example Workflow



Qyery Template (Q122, Stats): ❶

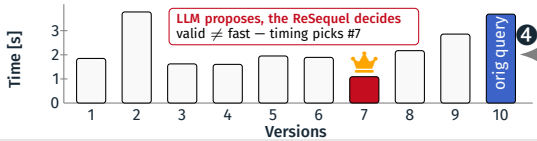
```
1:SELECT COUNT(*)
2:FROM History h,
3:  Posts p,
4:  Users u,
5:  Badges b
6:WHERE b.UserId = u.Id
7:  AND p.OwnerUserId = u.Id
8:  AND h.UserId = u.Id
9:  AND h.CreationDate >= '$$$'
10: AND h.CreationDate <= '$$$'
11: AND p.PostTypeId = ###;
```

Masked Predicates

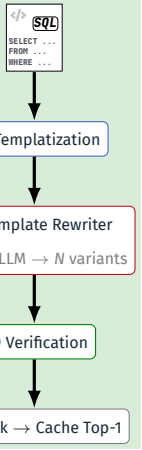
Output Template (Version #1):
Output Template (Version #2):
Output Template (Version #3):
Output Template (Version #4):
Output Template (Version #5):
Output Template (Version #6):

Output Template (Version #7):

```
1:WITH FilteredHistory AS (
2:  SELECT UserId, COUNT(*) h_count
3:  FROM History WHERE CreationDate >= '$$$'
4:    AND CreationDate <= '$$$' GROUP BY UserId),
5:FilteredPosts AS (
6:  SELECT OwnerUserId, COUNT(*) p_count
7:  FROM Posts WHERE PostTypeId = ###
8:  GROUP BY OwnerUserId),
9:FilteredBadges AS (
10:  SELECT UserId, COUNT(*) b_count
11:  FROM Badges GROUP BY UserId)
12:SELECT SUM(ph.ph_count * p.p_count * b.b_count)
13:FROM Users u
14:  JOIN FilteredHistory h ON u.Id = h.UserId
15:  JOIN FilteredPosts p ON u.Id = p.OwnerUserId
16:  JOIN FilteredBadges b ON u.Id = b.UserId;
```



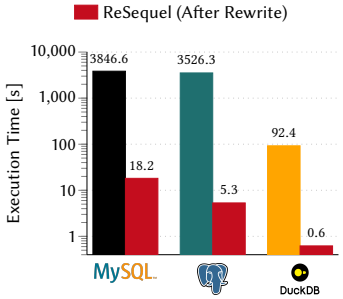
Rewriting pipeline:



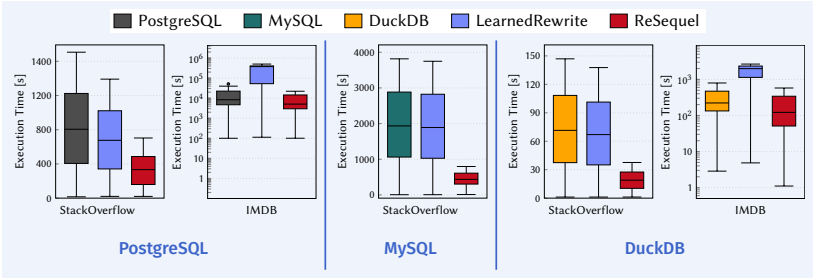
What ReSequel did to Q122

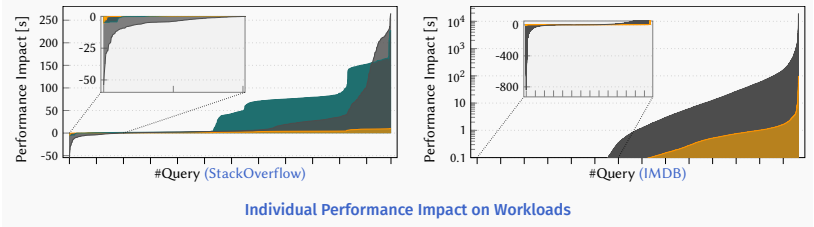
- **Factorized join** (eager aggregation) — `COUNT(*)` becomes a **product of per-key counts**; the flat join is never materialized
- **No 4-way cross join** — explicit joins on the CTEs; `Users` was only a PK bridge
- **Filters pushed into the CTEs** — applied *before* any join

148×–662× faster, same result



End-to-End Performance on Large Workloads





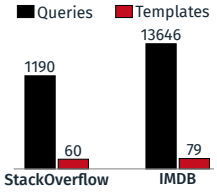
Individual Performance Impact on Workloads

Highlights:

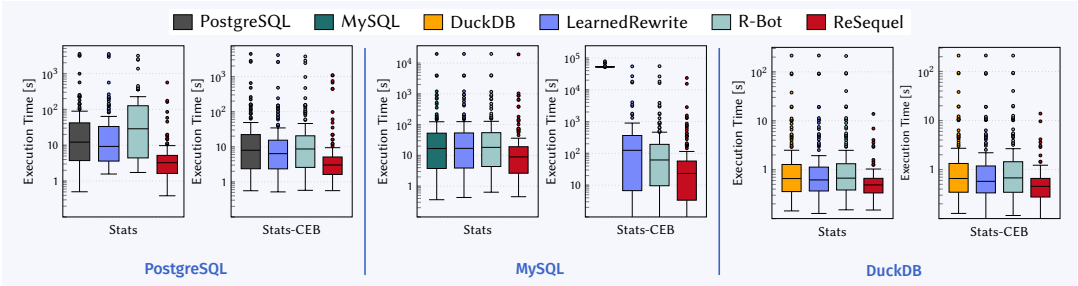
Workload speed-up

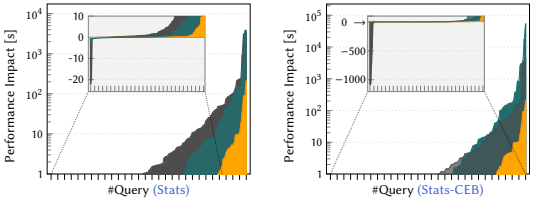
Host DBMS	StackOv.	IMDB
PostgreSQL	2.4×	2.3×
MySQL	4.8×	—
DuckDB	3.9×	1.4×

- **Every host DBMS improves** — not a single-engine effect
- **LLM baselines cannot run** — R-Bot, LLM-R2 rewrite *per query*
- **Per query:** most speed up



End-to-End Performance on Small Workloads





Individual Performance Impact on Workloads

Highlights:

Workload speed-up:

Workload	PostgreSQL	MySQL	DuckDB
Stats	5.89×	1×	14.85×
Stats-CEB	4×	3.3×	16×

- **Where the gain comes from**: up to **20 variants** per query, *data*/*catalog* split, redundant `GROUP BY` removed
- **R-Bot**: rewrites only **12%** of the workload — with **negative** impact
- **LearnedRewrite**: per-query, Calcite-bound — improves few queries, **often slowdowns**

Summary and Conclusions

- **Rule sets stay brittle** — phase ordering, rewrite budgets, strict pattern matching
- **Rewrite templates, not queries** — **scalable, reusable, robust**; pay the LLM once
- **Metadata guides the LLM** — correct rewrites, large runtime gains

16× over native DBMSs, **22**× over LLM rewriters, **600**× per query

Summary and Conclusions

Thank You!

- **Rule sets stay brittle** — phase ordering, rewrite budgets, strict pattern matching
- **Rewrite templates, not queries** — **scalable, reusable, robust**; pay the LLM once
- **Metadata guides the LLM** — correct rewrites, large runtime gains

16× over native DBMSs, **22**× over LLM rewriters, **600**× per query

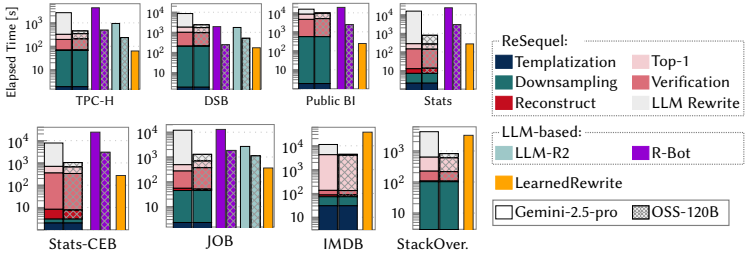
Workloads and Datasets

Dataset	#Tables	Workload	#Queries	#Templates
TPC-H	8	TPC-H (sf=10)	22	22
DSB	25	DSB (sf=100)	52	52
Public BI	83		100	100
Stats/Stats-CEB	8		146	146
JOB	21		113	95
StackOverflow	8	SQLStorm	1,192	60
IMDB	21		13,646	79

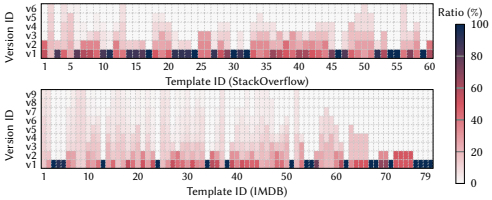
Used datasets and their workload characteristics

- **Group 1 — one template per query**
  TPC-H, DSB, Public BI, Stats/Stats-CEB: queries differ only in masked parameters, so templates = queries
- **Group 2 — partial overlap**
  JOB: ≈ 15% of queries share a template (113 queries, **95 templates**)
- **Group 3 — heavy overlap**
  StackOverflow and IMDB: 1,192 and 13,646 queries collapse to **60 and 79 templates**
- **Why the groups matter** — rewriting once *per template* only pays off when queries repeat; Group 3 is where that leverage is largest
- **Three hosts** — every workload runs on PostgreSQL, MySQL, and DuckDB

What Does Rewriting Cost?



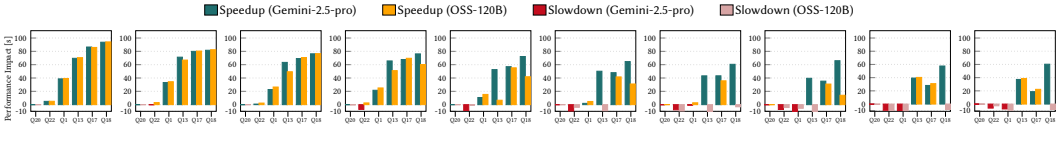
(a) Time breakdown of rewriting across 8 datasets (10% downsampling, PostgreSQL)



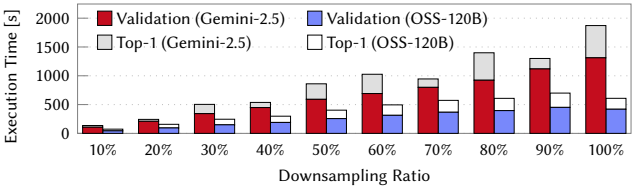
(b) Template and query cache hits

- **Templatization is free** — the catalog build dominates, not the LLM
- **LLM time scales with #templates**, not #queries (Gemini > OSS-120B, thinking mode)
- **Groups 1 & 2** — one template per query, so the cache never pays
- **Group 3** — **60 / 79** templates reused across 1,192 / 13,646 queries

How Much Data Does Verification Need?

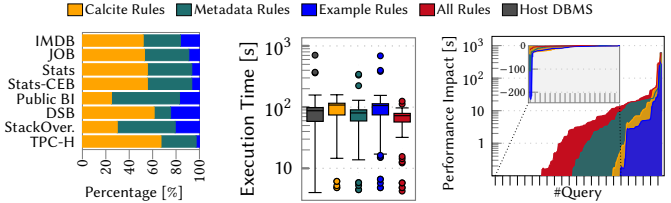


(a) Individual query performance impact at downsampling ratios 10%–100% (TPC-H SF10, PostgreSQL)



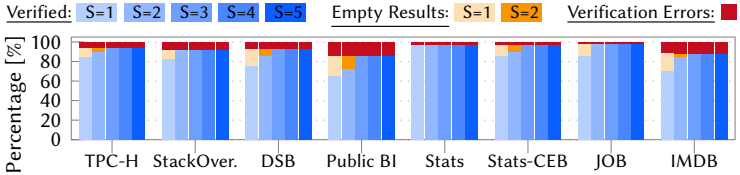
(b) Validation and top-1 selection overhead across downsampling ratios (TPC-H SF10, PostgreSQL)

Ablation Study



(a) Rule ratio · Overall (JOB) · Individual (JOB) — PostgreSQL

- **S=2–3** verifies every workload — more samples buy nothing
- All rules: **5.64×** on JOB Calcite **4.4×**, metadata **2×**
- Each category alone is erratic — the win is their overlap



(b) Verification rates across sample sizes (S=1–5; Gemini-2.5-pro, PostgreSQL)