Low Rank Learning for Offline Query Optimization

Zixuan Yi[1], Yao Tian[2], Zack Ives[1], Ryan Marcus[1]

[1] University of Pennsylvania [2] The Hong Kong University of Science and Technology

Simple, low-overhead **linear methods** can be almost as effective as complex **deep learning approach** for **Offline Learned QO**.

Offline Learned QO

Why? Current Learned QOs cause **unpredictable regressions**. ("my query was fast yesterday, why is it slow today?")

Observation: **Repetitive** workload!
Study at RedShift: **60%** of queries are repeated!

How? verify that potential new query plans are actually better than the default plan.

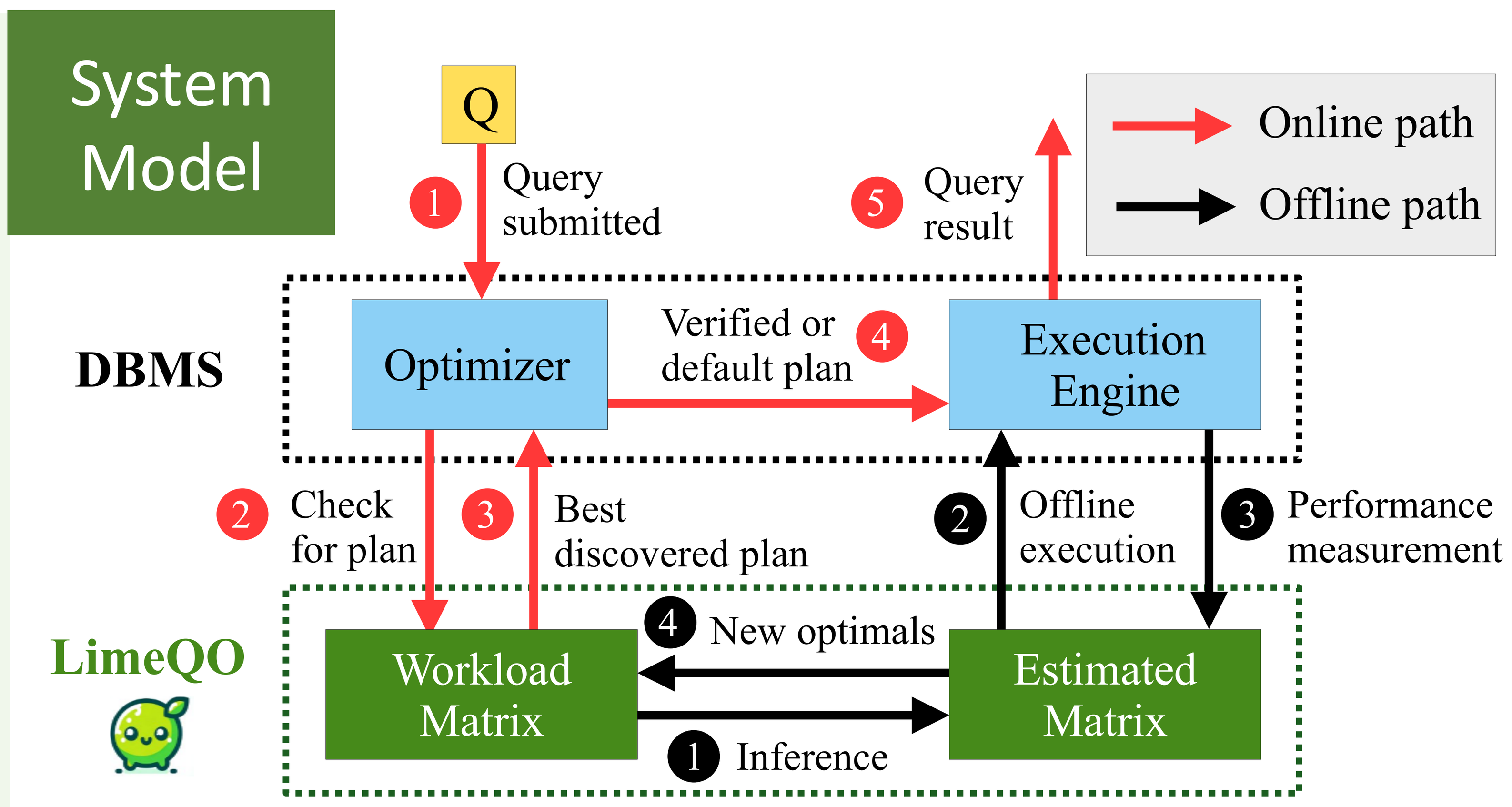
Goal: simultaneously minimize the workload latency and the total offline exploration time, while maintaining the "no-regressions" guarantee.

Low Rank Workload Matrix

Workload Matrix *M*:
Each row represents a SQL query.
Each column represents a hint (parameterization of the QO).
One possible hint: ***Disable*** Nested Loop Join, *Enable* Hash Join, *Enable* Merge Join, *Enable* Index Scan, *Enable* Seq Scan, *Enable* Index-only Scan
Each entry represents the latency time for DB to execute the query under the hint.

💡M is **LOW RANK**
Intuition: two queries that behave similarly on some hints are likely to behave similarly on other hints as well.

System Model

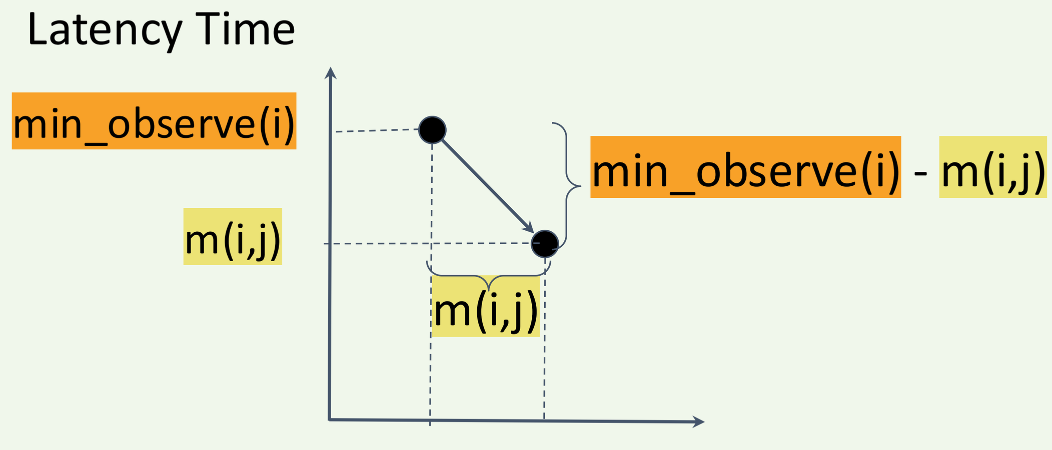


Active Learning on a Low Rank Matrix

Active Learning: intelligently select which new pieces of information to observe next
In our case: to reduce the exploration cost by revealing the most informative entries

For query i, I want to explore a new hint j

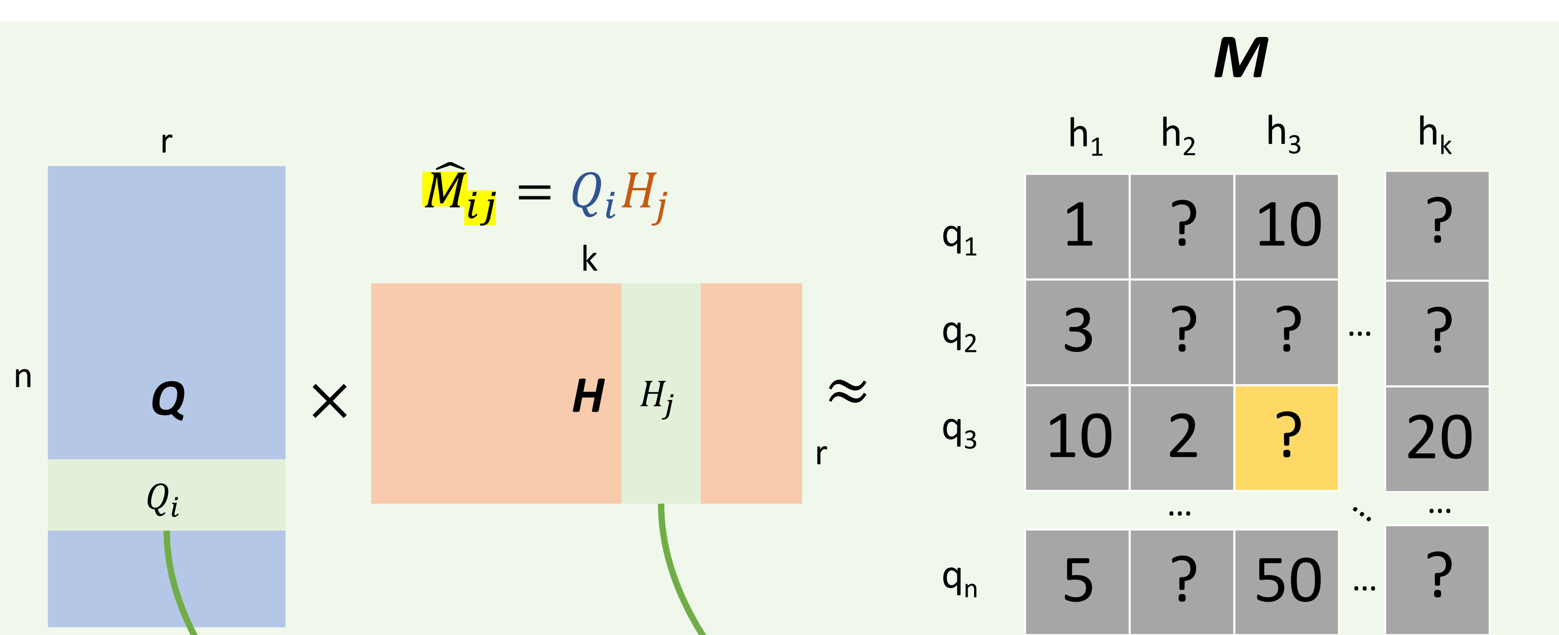
1 the new hint improves! ☺



Explore the queries with the **biggest improvement ratio:** (min_observe(i) – m(i,j))/m(i,j)
How to predict m(i,j)? **LimeQO** / **LimeQO+**

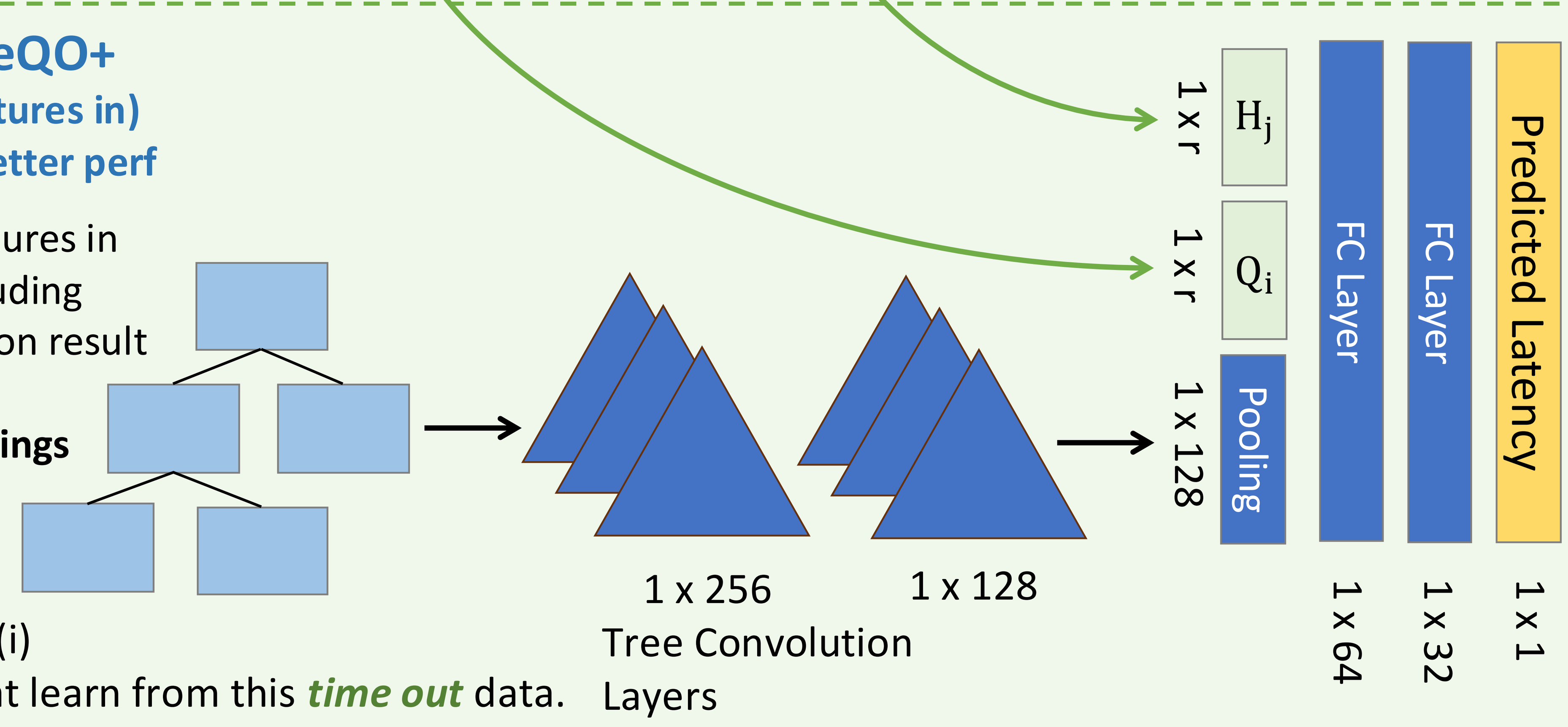
Option #1: LimeQO
(Linear Method Only)
want a low overhead solution

Use **Alternating Least Squares** Algorithm to recover the unobserved entries from the observed ones.



Option #2: LimeQO+
(Adding Query Features in)
higher compute, better perf

Use query plan features in tree structure (including cardinality estimation result and cost) and QH Matrix **embeddings** as input.

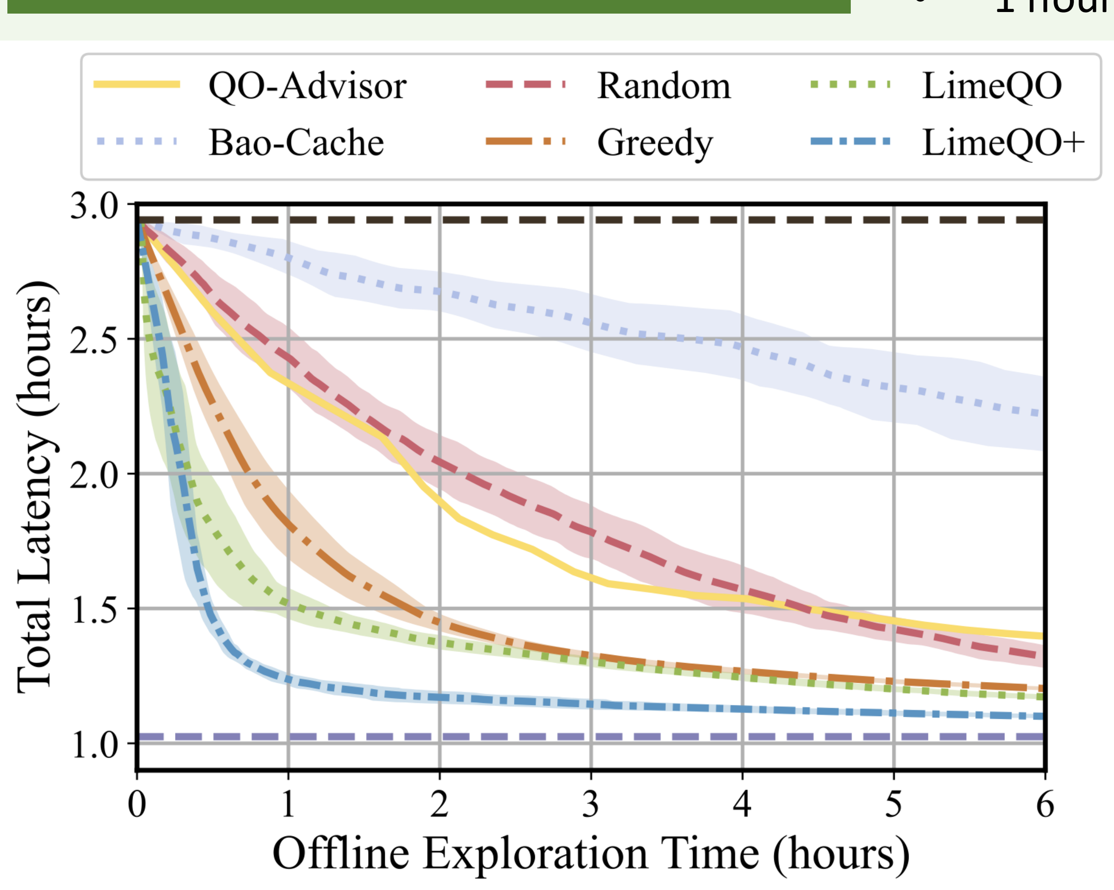


2 the new hint regresses ☹ meaning that m(i,j) is worse than min_observe(i)
It's okay! We timed out after min_observe(i) and use *censored* technique that learn from this *time out* data.

Experiments

Dataset: CEB core workload (3133 queries in total, 49 hints)

- takes ~3 hours for PostgreSQL default to finish
- ~1 hour if every query is chosen the optimal hint



QO-Advisor select the unexplored entry with the lowest optimizer cost.
Bao-Cache the technique of Bao adapted to offline exploration. Cache the results to guarantee no regression.
Random randomly explore unobserved entries.
Greedy explore the tail latency queries first.
LimeQO uses only Linear Method to predict.
LimeQO+ uses query features and matrix embeddings to train and predict.

Total Latency Time (workload latency) is simply adding up the observed row minimum in the workload matrix.
Offline Exploration Time is the total time to execute the query plan + overhead time of the technique.

Caption: Both LimeQO and LimeQO+ outperform baselines.
Even without any features, pure linear method (LimeQO) can perform nearly as effective as the one using complex Neural Network (LimeQO+).

Overhead Time comparison: LimeQO's overhead time is only 10 seconds, while LimeQO+ takes 1 hour (1/6 of the offline exploration).

Checkout the paper for more detailed info ⬇

Code ⬇

My website ⬇