Lecture #4: Towards Robust Query Execution

15-721 Advanced Database Systems (Fall 2024)

https://www.cs.cmu.edu/~15721-f24//

Carnegie Mellon University
Author: Karthik Balaji Ganesh

1 Introduction

Query Optimization is a critical component of any Database Management System (DBMS) that is responsible for generating an efficient (and ideally optimal) query execution plan. Queries are typically written in a high-level declarative language such as SQL and then parsed, and the query optimizer is responsible for exploring the vast search space of possible plans to identify an efficient execution strategy.

Since this is a vast search space, the query optimizer might employ different cost estimations, heuristics and statistics to prune the search space and identify the best plan. One such heuristic is **cardinality estimation**, particularly in the case of joins. While accurate cardinality estimation is critical in determining an optimal join order, in practice, it is observed that query optimizers make cardinality estimation errors. Crucially, these estimation errors grow exponentially with successive joins, as observed in Figure 1 and Figure 2. This seems like a natural expectation, due to the propagation of these errors in each successive join.

The Zipfian distribution seen in Figure 1 is a common distribution in real-world data, where the data is very likely to be skewed. Additionally, in Figure 2, we can observe that all the errors are below the expected line; in other words, all the database systems are prone to underestimating the join size, which is disastrous for estimating join costs, and overall query optimization.

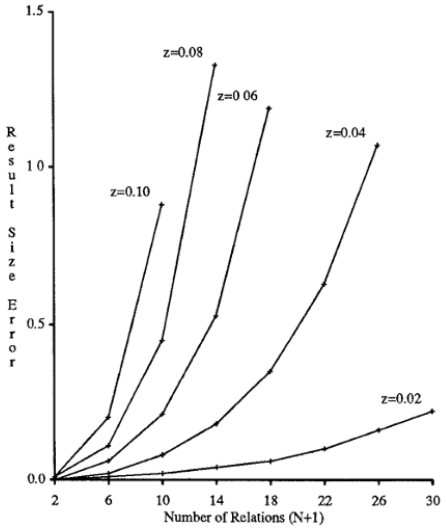

Figure 1: Join result size error for Zipf distributions under uniform approximation

Mitigating these errors becomes a pressing issue in the modern disaggregated world, where the data is stored in vast lakehouses far away from the compute resources. The consequence of this is that, at runtime, the database system often has no or minimal statistics of the underlying data, and runtime optimization becomes a necessity. Thus, it becomes extremely important to perform query execution in a robust manner at runtime, such that even a suboptimal query plan proposed by the optimizer is not too far away from the optimal query plan in terms of runtime performance. It is even more beneficial if these techniques

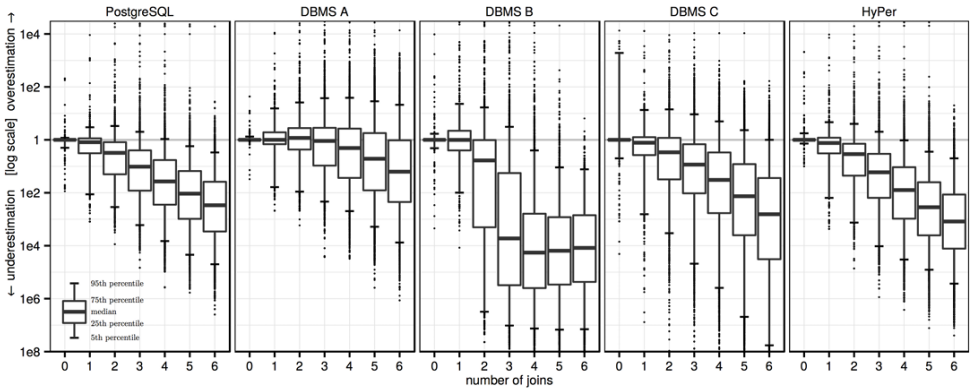

Figure 2: Estimation errors by different public DBMSs

can be shown to be provably robust. The focus therefore shifts to defining this notion of robustness, and subsequently exploring techniques that achieve this robustness in query execution, i.e., execution techniques that are immune to a poor plan selection. One such technique that will be explored in detail is **Lookahead Information Passing (LIP)**.

2 Lookahead Information Passing

2.1 Setting

The setting is limited to the star schema for the next few sections. In a star schema, there is a fact table at the center, and multiple dimension tables surrounding it. The fact table is typically large, and the dimension tables are small. The objective is to inner-join the fact table with the dimension tables; for a star schema, the optimal join plan is a left-deep plan.

2.2 Semi-join Optimization

A **semi-join** is a join operation that is similar to an inner join, except that it only returns columns from the left-side table, while also avoiding duplicates. While semi-joins are a useful optimization in themselves, they can be optimized even further by employing a filter, typically a bloom filter.

In this scenario, while the hash table is being constructed on the build side, the filter is also constructed alongside in the same pass. This filter is then passed to the probe side. This allows the probe side to use the filter as a preliminary sieve, to avoid probing the hash table for tuples that are not present in the filter. In other words, this trades off the cost of probing the hash table against the cost of probing the filter. This is a classic example of a semi-join optimization, and is also known as **sideways information passing**.

2.3 Filter Data Structure

The primary purpose of leveraging the filter, as discussed in the previous section, is to use it as a cheaper, in-memory alternative to probing the hash table and offset the overall per-tuple cost. As a result, this filter should be designed to be as efficient as possible. Typically, the data structure that is commonly used is either a **bloom filter** or a **bitvector**.

An in-memory bloom filter is compact, occupies very little memory, and is thus more likely to be cache resident. False positives are not too expensive, and the size of a bloom filter is typically a tradeoff against the desired false positive rate, i.e., the required size can be computed based on the number of unique values, and the desired false positive rate. For n inserted elements, and a desired false positive probability of ϵ, the required number of bits can be computed as $m = -\frac{n \ln \epsilon}{(\ln 2)^2}$. These parameters must thus be tuned accordingly, to ensure that the bloom filter lookup is actually more efficient than a hash table probe. While the number

of hash functions is another parameter that is usually tuned, in most cases the optimal number is 1 in an in-memory setting. In the case of the star schema, the bloom filters are all likely to be small and in-memory, and thus cache resident, due to the small size of the dimension tables.

A bitvector is typically used in scenarios where you know you have a very small number of unique values, and one bit is used to represent each unique value.

2.4 LIP Algorithm

With Lookahead Information Passing, the discussed semi-join optimization is scaled to multiple joins in a very efficient manner, and is summarized below.

Algorithm 1 Lookahead Information Passing

1: For each dimension table in the join, a filter is constructed alongside the build-side hash table.
2: All these filters are then passed to the fact table on the probe side.
3: To mitigate the cost of expensive hash table probes, the fact table applies **all the filters** before probing any hash table.

It can be observed that the overall cost of probing the filters would depend on the order in which the fact table applies the filters. In Figure 3, if $\sigma(D_1)$ is more selective than $\sigma(D_2)$, then applying the former first would result in a reduced number of tuples in the fact table before the second filter is applied, thereby minimizing the overall number of filter probes and the amortized per-tuple cost.

Thus, the goal is to apply the filters in the order of their selectivity. However, this once again seemingly reduces to a cardinality estimation problem, which has been determined to possess an exponential number of errors. The alternate solution is to use an **adaptive reordering** technique.

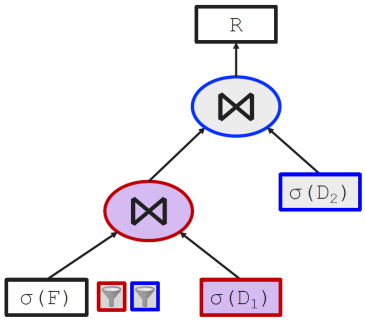

Figure 3: Overview of the LIP algorithm tree with 2 joins

2.5 Adaptive Reordering

The adaptive reordering algorithm is a simple, yet effective runtime heuristic that is employed to determine the order in which the filters are to be applied. The algorithm is as follows:

Algorithm 2 Adaptive Reordering

1: Construct a batch of fact-side tuples from the fact table (anywhere from 64 to 1024 tuples).
2: Apply the first filter, and update the hit and miss statistics for this filter.
3: Apply the next filter for all tuples that have passed through the first sieve. Repeat this process until all the filters have been applied.
4: Sort the filters in descending order by miss rate i.e. the most selective filter comes first.
5: Repeat the process for the next batch of fact-side tuples. This batch size is also adaptively increased, typically at a 2x rate per step.

Thus, this runtime adaptive reordering algorithm is used to determine the order in which the filters are to be probed. Using batching improves the overall cache efficiency of the filter probes; this is similar to how vectorized query execution is more cache-friendly than an iterative model. Smaller batches allow faster adaptation, since the filters are rearranged more frequently, while larger batches ensure greater cache efficiency due to the contiguous nature of the tuples that are being processed. Thus, the batch size is adaptively increased as we progress in the algorithm, accounting for our increased confidence in the filter order as the sample size of our hit/miss statistics increase. However, make sure to ensure that your batch sizes remain cache conscious, as very large batch sizes may overflow out of the cache.

Practically, it has been observed that this adaptive filtering in fact converges to the **optimal join order**. This has a massive implication: there is no requirement to optimize an equijoin subtree with a left-deep structure, since this adaptive filtering algorithm ensures its practical optimality.

2.6 Benefits of LIP

- Once all the filters have been applied, the number of fact table tuples is on the same order of magnitude as the number of tuples in the final result. The precise numbers would depend on the false positive rates. As a consequence, each hash table is only probed a minimal number of times.
- Adaptive reordering ensures that the number of lookahead filter probes is roughly optimal too, by utilizing the selectivity of these filters as an effective heuristic.
- Consequently, Lookahead Information Passing guarantees to quickly converge to an optimal join order, which not only alleviates the burder on the query optimizer, but ensures that a near-optimal join performance can be achieved even with a suboptimal query plan.

3 Theoretical Results

3.1 Robustness

- Robustness is defined in terms of the difference in costs between the best and worst plans for any strategy.
- This difference is normalized by the spread of selectivities ($\sigma_{\max} - \sigma_{\min}$) and the size of the fact table. This ensures that the robustness is independent of the scale of the query. It also indicates that in general, the propensity of the optimizer to make errors increases with the spread in selectivites, as well as the size of the fact table, which are natural expectations.
- A more rigorous definition of Θ-robustness is provided in Figure 4. When $\sigma_{\max} - \sigma_{\min} = 0$, there is no join problem to solve, as every join is equally selective.

DEFINITION 1. *An evaluation strategy $\mathcal{E}$ is said to be θ-fragile and Θ-robust and with respect to a plan space $\mathcal{P}$ if the maximum deviation in performance of any plan in $\mathcal{P}$ (including the worst plan $\mathcal{E}_w$) from the best one $\mathcal{E}_b$, normalized by the fact table cardinality and spread of selectivities in a query, is bounded between θ and Θ.*

$$\theta \leq \frac{T(\mathcal{E}_w) - T(\mathcal{E}_b)}{(\sigma_{max} - \sigma_{min})|F|} \leq \Theta, \qquad \sigma_{max} \neq \sigma_{min} \tag{9}$$

Figure 4: Definition of Θ-robustness and θ-fragility

3.2 Robustness Bounds - General Case

Given selectivities $\sigma_{\min} = \sigma_{1'} \leq \sigma_{2'} \leq ... \leq \sigma_{n'} = \sigma_{\max}$, for a particular permutation $P_{12...n}$, it can be observed that on the first step when joining with D_1, the build-side hash table is probed $|F|$ times; next, on joining with D_2, the fact table is probed $\sigma_1|F|$ times, and so on. The total hash table probing cost of this permutation is thus $\sum_{i=1}^{n} \sigma_0...\sigma_{i-1}|F|$. Here, for ease of representation, $\sigma_0 = 1$.

Now, we can obtain upper and lower bounds for this cost, using the observation that for all i, $\sigma_{\min} \leq \sigma_i \leq \sigma_{\max}$. Subsequently,

$$\begin{aligned} \sum_{i=1}^{n} \sigma_{\min}^{i-1}|F| &\leq \sum_{i=1}^{n} \sigma_0...\sigma_{i-1}|F| \leq \sum_{i=1}^{n} \sigma_{\max}^{i-1}|F| \\ \implies \frac{1-\sigma_{\min}^{n-1}}{1-\sigma_{\min}}|F| &\leq \sum_{i=1}^{n} \sigma_0...\sigma_{i-1}|F| \leq \frac{1-\sigma_{\max}^{n-1}}{1-\sigma_{\max}}|F| \end{aligned}$$

The above bounds are directly obtained from the formula for the sum of a geometric progression with n elements, starting element 1, and ratio $\sigma_{\min}$ and $\sigma_{\max}$ respectively. These are thus the bounds on the hash table probing cost of *any* plan.

We can now extend a similar idea to compute the difference between the best (b) and worst (w) plans. This is based on the observation that the join order in the best and worst plans will be the opposites of each other. Using these ideas,

$$\begin{aligned} T(P_w) - T(P_b) &= \sum_{i=1}^{n-1} \left(\sigma_{n'}...\sigma_{(n-i+1)'} - \sigma_{1'}...\sigma_{i'}\right)|F| \\ &= \sum_{i=1}^{n-1} \left(\sigma_{n'}\left(\sigma_{(n-1)'}...\sigma_{(n-i+1)'}\right) - \sigma_{1'}\left(\sigma_{2'}...\sigma_{i'}\right)\right)|F| \\ &\geq \sum_{i=1}^{n-1} \left(\sigma_{n'}\sigma_{1'}^{n-1} - \sigma_{1'}\sigma_{1'}^{n-1}\right)|F| \\ &= \sum_{i=1}^{n-1} \left(\sigma_{n'} - \sigma_{1'}\right)\sigma_{1'}^{n-1}|F| \\ &= \sum_{i=1}^{n-1} \left(\sigma_{\max} - \sigma_{\min}\right)\sigma_{\min}^{n-1}|F| \\ &= \frac{1-\sigma_{\min}^{n-1}}{1-\sigma_{\min}}\left(\sigma_{\max} - \sigma_{\min}\right)|F| \end{aligned}$$

From our earlier definition of θ-fragility, we can observe that, $\frac{T(P_w)-T(P_b)}{(\sigma_{\max}-\sigma_{\min})|F|} \geq \frac{1-\sigma_{\min}^{n-1}}{1-\sigma_{\min}} \implies$ non-LIP (traditional) query plans are θ-fragile with respect to the space of left-deep plans for $\theta = \frac{1-\sigma_{\min}^{n-1}}{1-\sigma_{\min}}$

3.3 Robustness Bounds - Adaptive Reordering

Now, a similar analysis can be performed for the adaptive step in the LIP algorithm to derive an upper bound instead of a lower bound i.e. Θ-robustness can be demonstrated.

If the cost of probing a Bloom filter is assumed to be a constant β, and if a false positive rate of ϵ is assumed, the overall cost of probing for the first filter would be $\beta|F|$, for the second filter would be $(\sigma_{1'} + \epsilon)\beta|F|$ and so on. Thus, the total cost of probing the bloom filter, for a plan $P_{12...n}$, is,

$$\text{BloomFilterProbeCost}(P_{12..n}) = \left(1 + \sum_{i=1}^{n-1}(\sigma_{1'} + \epsilon)...(\sigma_{i'} + \epsilon)\right)\beta|F|$$

Observe here that the optimal selectivity order is used in the computation of the bloom filter probe cost, since it is independent of the actual join order.

Now to compute the cost of probing the hash table, we make the observation that at any given join, the differentiating factors from the previous joins would be the number of false positives weeded out. That is, the first join's hash table probe cost would be $(\sigma_1 + \epsilon)...(\sigma_n + \epsilon)|F|$, but the second join's probe cost would only be $\sigma_1(\sigma_2 + \epsilon)...(\sigma_n + \epsilon)|F|$ due to the false positive removed, and so on. Thus,

$$\begin{aligned}\text{HashTableProbeCost}(P_{12..n}) &= \sum_{i=1}^{n} \sigma_0\sigma_1...\sigma_{i-1}(\sigma_i + \epsilon)...(\sigma_n + \epsilon)|F| \\ &\simeq \sigma_1...\sigma_n|F| \left(\sum_{i=1}^{n} 1 + \epsilon\left(\frac{1}{\sigma_i} + ... + \frac{1}{\sigma_n}\right)\right)\end{aligned}$$

In the above computation, higher-order ϵ terms are assumed to be negligible. We can clearly observe that the overall hash table probe cost is minimized when the join is performed in the order of $\sigma_{1'} \leq \sigma_{2'} \leq ... \leq \sigma_{n'}$ i.e. arranging the filters from most selective to least selective indeed obtains the optimal join order, and also minimizes the total cost. This reaffirms our empirical observation that arranging the filters according to their selectivities helps converge to the optimal join order and best performance. Now, similar to our previous analyses, we can use $\sigma_{\min}$ and $\sigma_{\max}$ in the above equations to derive bounds as,

$$\sigma_1...\sigma_n|F|\left[n + \frac{\epsilon}{\sigma_{\max}}\frac{n(n+1)}{2}\right] \leq \text{HashTableProbeCost}(P_{12..n}) \leq \sigma_1...\sigma_n|F|\left[n + \frac{\epsilon}{\sigma_{\min}}\frac{n(n+1)}{2}\right]$$

Observe that for both the best and worst plans, with LIP, the build costs and bloom filter probe cost will remain the same. Thus, the difference in costs only depends on the hash table probe costs, and the above bounds can be used to derive that,

$$\begin{aligned}T(P_w) - T(P_b) &\leq \frac{1}{2}\sigma_1...\sigma_n n(n+1)\left[\frac{1}{\sigma_{\min}} - \frac{1}{\sigma_{\max}}\right]|F| \\ \implies \frac{T(P_w) - T(P_b)}{(\sigma_{\max} - \sigma_{\min})|F|} &\leq \frac{1}{2}\frac{\sigma_1...\sigma_n}{\sigma_{\min}\sigma_{\max}}n(n+1)\end{aligned}$$

Thus, LIP with adaptive reordering is Θ-robust for $\Theta = \frac{1}{2}\frac{\sigma_1...\sigma_n}{\sigma_{\min}\sigma_{\max}}n(n+1)$. In other words, for left-deep inner join subtrees, any plan is close to the optimal plan with adaptive LIP, and we have obtained a provable bound on the difference. What this bound also shows is that, assuming all $\sigma_i < 1$, the value of Θ scales down dramatically as the number of joins increases, i.e., with more joins, the bound becomes tighter and tighter, and thus, the harder problems actually become simpler.

4 Experimental Results

In Figure 5, 6, and 7, we can see the experimental performances of LIP in different settings and benchmarks.

5 Moving Beyond the Star Schema

5.1 The Yannakakis Algorithm

The Yannakakis algorithm moves beyond the star schema, and asserts that for *any* acyclic conjunctive query, the query can be evaluated in polynomial time w.r.t. the size of the database. It does this by utilizing the join graph, as seen in Figure 8.

The algorithm begins by picking up the graph at any node in the graph. The choice of node can be arbitrary, but would impact the final runtime of the algorithm. Say for example, that we pick S. First, an upward

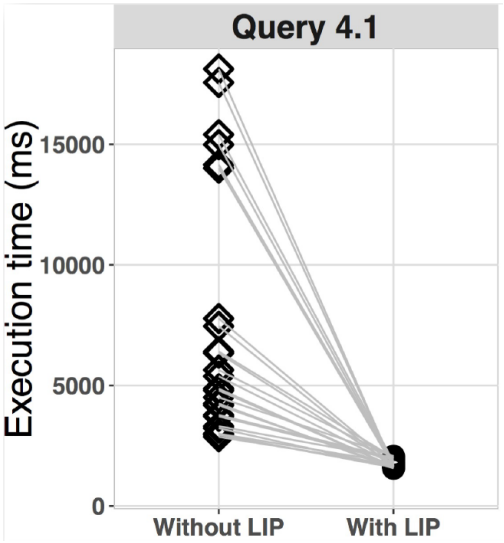

Figure 5: Star Schema Benchmark at SF 100 (100 GB)

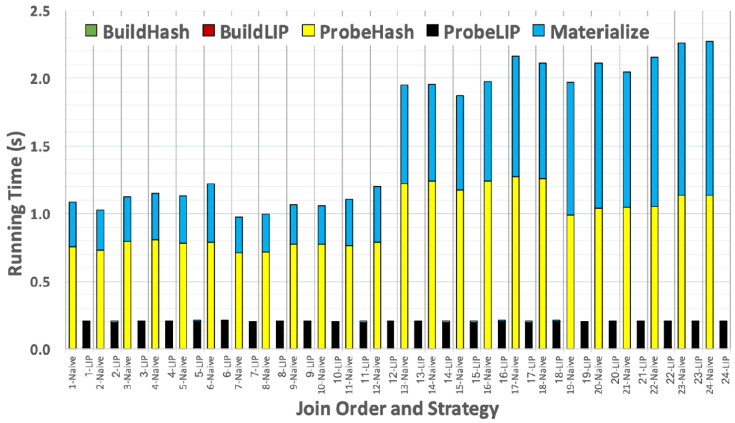

Figure 6: Cost breakdown for Star Schema Benchmark

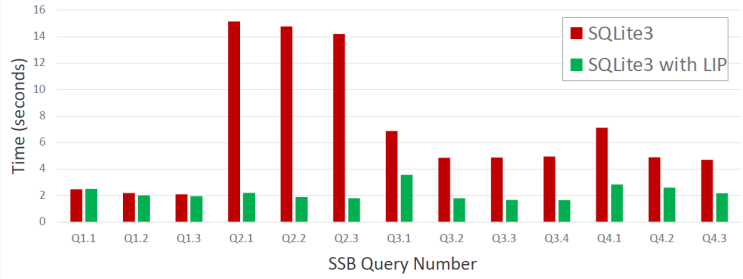

Figure 7: Performance in SQLite3

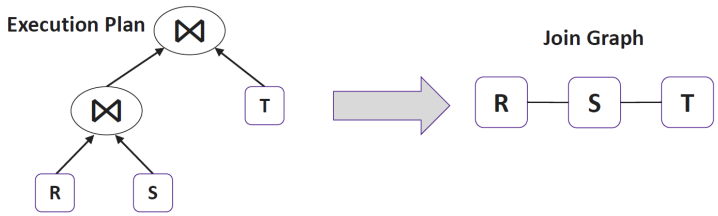

Figure 8: Converting an execution plan to a join graph

pass is performed, moving upwards from the children, wherein the parent is filtered to only include tuples that match with its children. This is performed till we reach the root. For example, in this case, S is filtered with R's tuples to obtain S', which is in turn filtered with T's tuples to obtain S''.

After this, starting at the root, an inverse downward pass is performed, where now each child is filtered to only include tuples that match with tuples from its parent. Thus, R is filtered with tuples from S'' to obtain R', and T is filtered with tuples from S'' to obtain T'. This process is repeated till we reach the leaves.

Finally, the join phase is performed, on these filtered relations i.e. the new join graph is as seen in Figure 9. This join graph contains heavily filtered relations, and the query can be evaluated in polynomial time w.r.t. the size of the database.

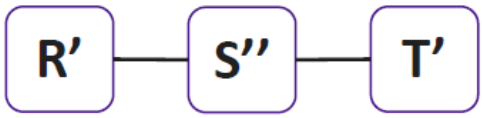

Figure 9: The final join graph after applying the Yannakakis algorithm

5.2 Query Optimization vs Adaptive Query Processing

ML-based Query Optimization is a field that has recently seen a great surge in interest. Figure 10 compares Balsa, an RL-based query optimizer, against PostgreSQL. It can be seen that an overfitted model (a model trained on both the training and test data) is faster than PostgreSQL, but the general model (trained only on the training data) is worse in comparison. In other words, it appears that the ML-based query optimizers are great at memorizing patterns efficiently, but they do not seem to truly learn and are unable to generalize to new patterns and workloads. This can also be seen in Figure 11, where Balsa and Bao (another ML-based optimizer) are both outperformed by adaptive QO.

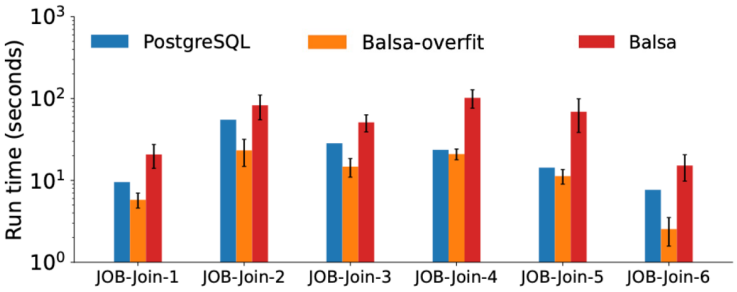

Figure 10: Balsa (ML-based QO) vs PostgreSQL

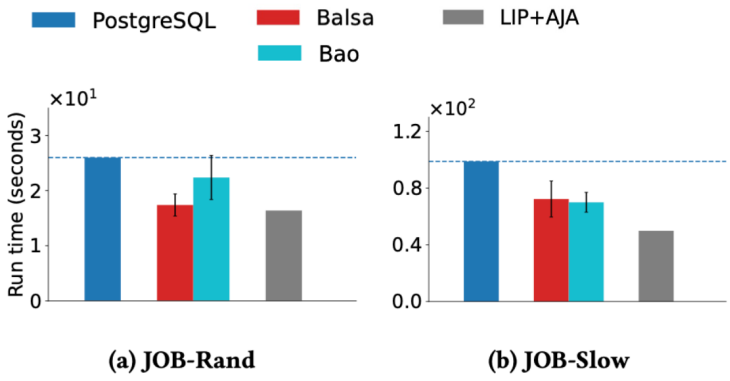

Figure 11: ML-based QO vs Adaptive QO

6 Conclusion

- While query optimization has always been and will continue to remain a cornerstone of relational databases and efficient data querying, we have now observed that adaptive query planning is also an important future direction in building a robust and efficient system.
- Lookahead Information Passing (LIP) is one such adaptive query planning algorithm that is both simple to implement as well as provably robust. In the case of left-deep inner join trees in an in-memory setting with the star schema, LIP results in optimal run-time behavior.
- Due to its simplicity and efficiency, LIP has also been integrated into commercial database systems such as SQLite3.
- Future works can look to extend this technique and this general class of adaptive algorithms to other query plan shapes, other schema types, as well as distributed settings.