Looking Ahead Makes Query Plans Robust

Making the Initial Case with In-Memory Star Schema Data Warehouse Workloads

Jianqiao Zhu Navneet Potti Saket Saurabh Jignesh M. Patel
Computer Sciences Department
University of Wisconsin–Madison
{jianqiao, nav, saket, jignesh}@cs.wisc.edu

ABSTRACT

Query optimizers and query execution engines cooperate to deliver high performance on complex analytic queries. Typically, the optimizer searches through the plan space and sends a selected plan to the execution engine. However, optimizers may at times miss the optimal plan, with sometimes disastrous impact on performance. In this paper, we develop the notion of robustness of a query evaluation strategy with respect to a space of query plans. We also propose a novel query execution strategy called *Lookahead Information Passing (LIP)* that is robust with respect to the space of (fully pipeline-able) left-deep query plan trees for in-memory star schema data warehouses. LIP ensures that execution times for the best and the worst case plans are far closer than without LIP. In fact, under certain assumptions of independent and uniform distributions, any plan in that space is theoretically guaranteed to execute in near-optimal time. LIP ensures that the execution time for every plan in the space is nearly-optimal. In this paper, we also evaluate these claims using workloads that include skew and correlation. With LIP we make an initial foray into a novel way of thinking about robustness from the perspective of query evaluation, where we develop strategies (like LIP) that collapse plan sub-spaces in the overall global plan space.

1. INTRODUCTION

Relational database management systems (RDBMSs) have a unique internal organization where query execution can be viewed as a composition of basic relational algebraic (RA) operations. This underlying framework allows RDBMSs to navigate the space of equivalent RA compositions to find the most efficient execution plan. This ability to optimize query plans is crucial to the RDBMSs' ability to execute complex queries efficiently even on large databases.

Query optimization, however, is a complex task. Decades of research in this area have yielded a plethora of techniques for plan enumeration, cardinality and cost estimation, and dynamic query optimization. Despite these remarkable advancements, it is well known [17, 23] that query optimizers still falter in some cases, producing query plans that have disastrously worse performance than optimal.

Rather than directly improving the capability of query optimizers, in this work, we take an approach that is complementary to most prior work in query optimization. The question we seek to address is: *Can we develop query execution techniques that dramatically reduce the impact of a poor choice of a query plan?* Thus, the big picture view of our approach is to focus on developing efficient *query evaluation techniques* that increase the *robustness* of query plans, by mitigating issues related to bad plan selection within a subspace of plans. We further limit our scope to increasing the robustness of plans to errors in join order selection.

Lookahead Information Passing (LIP), the query evaluation strategy that we propose in this work, is targeted at the common scenario of star schema data warehouses. In such workloads, a natural space of "good" plans for a query optimizer is that of fully pipeline-able left-deep join trees. For instance, consider Query 4.3 in the Star Schema Benchmark, shown in Figure 2. Figures 2a and 2b show two of the 24 possible left-deep query plans for this query, resulting from all permutations of the 4 dimension tables in the query.

If the optimizer selects a poor join order for such a query, the intermediate join results will be needlessly large, incurring additional processing time for extraneous tuples. One approach to reducing the impact of bad plan selection, therefore, is to efficiently pre-filter such extraneous tuples. This idea underlies our proposed LIP strategy.

In essence, the LIP strategy consists of two components. First, we pass succinct filter data structures (such as Bloom filters) from the "outer" (dimension) relations in all the joins to the "inner" (fact) relation. Thus, we can approximately pre-filter the fact table before performing the join opera-

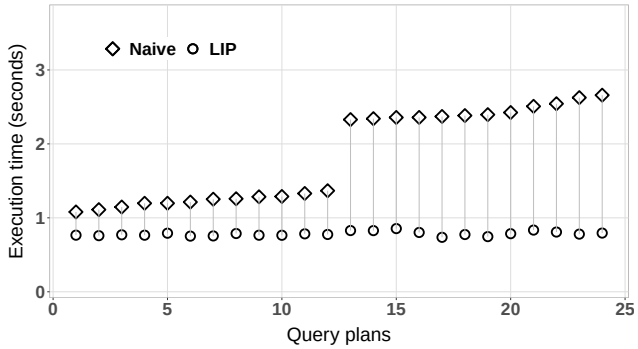

Figure 1: All 24 possible left-deep query plans for SSB Query 4.3 in increasing order of execution time.

Proceedings of the VLDB Endowment, Vol. 10, No. 8

```
SELECT d_year, s_city, p_brand1,
       SUM(lo_revenue - lo_supplycost) AS profit1
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_partkey = p_partkey
  AND lo_orderdate = d_datekey
  AND c_region = 'AMERICA'
  AND s_nation = 'UNITED STATES'
  AND (d_year = 1997 OR d_year = 1998)
  AND p_category = 'MFGR#14'
GROUP BY d_year, s_city, p_brand1
ORDER BY d_year, s_city, p_brand1;
```

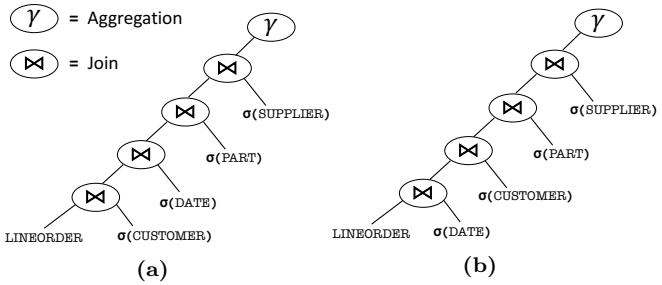

Figure 2: Query 4.3 from Star Schema Benchmark and two left-deep query plans for it.

tion. For the case of a hash join, for instance, this optimization can trade off expensive hash table probes in DRAM for more efficient filter probes in the CPU cache. Further, it also reduces the cost of materializing intermediate results in a vectorized query execution engine. Second, to minimize the number of filter lookups required, we use an adaptive reordering algorithm that dynamically converges to the optimal ordering for filter application.

Figure 1 illustrates how the use of LIP increases robustness in query processing. The execution times of the naive evaluation strategy (without using our proposed techniques), marked using diamonds, can be anywhere from 1.1s to 2.7s, depending on the join order selected. On the other hand, the execution times using LIP are all within 0.1s of each other. Further, not only does LIP have negligible overhead, we actually see an improvement in performance for typical queries. For instance, in 8 of the 13 SSB queries, the query plan using the LIP technique had an execution time better than the best query plan without using LIP.

A salient aspect of the LIP strategy is that its implementation requires only minimal changes in an existing query optimizer and execution engine. The technique is agnostic to the cardinality estimation and the plan selection methods that are used by the query optimizer. Further, the build and probe of these filter structures can be folded into the respective hash table operations in the join implementation, thus avoiding the need for any new operators or changes to the control flow in the execution engine.

The idea of passing a filter between the two sides of a join operation in the LIP technique bears resemblance to the well-known semi-join optimization. However, to the best of our knowledge, ours is the first work to aggressively use this optimization across multiple joins in a join tree. For this reason, the benefit of semi-join optimization for robust query processing has not been noted in literature. It is also important to note that the adaptive reordering phase of the LIP technique is crucial for both reducing sensitivity to estimation errors as well as speeding up performance.

In this work, we introduce a novel approach to robust query processing: one that focuses on query execution techniques that are immune to poor choices made by the query optimizer. In this initial foray, we have limited the scope to star schema data warehouses and left-deep query plans. LIP is also applicable to left-deep join tree subplans within larger query plans, as we demonstrate (in Section 5.6) using an example query from the TPCH benchmark.

While we admittedly address a simple scenario in this initial paper on robustness of query execution strategies, we are able to present elegant analytical results that we find at once insightful and intuitive. We believe that these insights will guide future research into robust query processing strategies for more complex scenarios.

We now summarize the contributions made in this paper.

1. A formal definition for the robustness of a query evaluation strategy, along with a simple analytical model that allows us to derive closed-form results about performance and robustness in a plan space.
2. A specific strategy, Lookahead Information Passing, that dramatically increases robustness to errors in join order selection, with little overhead (and often, improved performance), for the case of left-deep query plans in a star schema data warehouse.
3. Theoretical guarantees (based on the simple models) for the claims about robustness and near-optimality.
4. Some notes about implementation issues that had to be addressed in order to apply these techniques in a high-performance main memory analytics database system.
5. Empirical evaluation of LIP, including a new synthetic data and query generator that allows us to study the impact of skew and correlation on execution times.

2. PRELIMINARIES

We begin this section by defining the star schema. We scope the analysis in this paper to the plan space consisting of left-deep join trees for select-project-join (SPJ) queries in such a schema, operating in an in-memory setting. Then, we use a cost model to evaluate the performance of plans in this space, without using LIP. The results from this baseline analysis allow us to formally define the notion of robustness that we use in the rest of the paper. Finally, we provide a brief introduction to Bloom filters, the key data structure used in our implementation of LIP.

2.1 Star Schema and Left-deep Join Trees

Data warehouses used for decision support systems often follow the Kimball method [15] which results in a *star schema*. In this paper, we focus on this important pattern.

A star schema consists of a *fact* table F, often containing information about events such as sales and shipments, as well as a set of N *dimension* tables $\{D_1, D_2, \ldots, D_N\}$, containing additional descriptive attributes such as details about customers or products. The dimension tables are typically orders of magnitude smaller than the fact table, and are related to it through *primary key - foreign key* constraints.

We limit our focus to SPJ queries involving some n dimension tables, as shown in the RA expression:

$$F \quad \bowtie \quad \sigma(D_1) \quad \bowtie \quad \sigma(D_2) \quad \bowtie \quad \ldots \quad \bowtie \quad \sigma(D_n) \quad (1)$$

All the notation used throughout this paper is summarized in Table 1. Note that for notational convenience, we have dropped the selection predicate on F, though our results do not depend on this assumption.

Given typical table cardinalities in star schema warehouses, the optimal query plan for such a query likely uses the fact table as the "outer" relation in every join (i.e., the probe side of hash joins). We can visualize such a plan as a left-deep tree of joins. Such left-deep trees allow for full pipelining of the query results and are generally the optimal plan shape for queries of this type. We further scope the paper to only examine query plans with this shape.

EXAMPLE 1. *The Star Schema Benchmark (SSB) [22] is a widely used variation of the TPC-H benchmark, and is a star schema data warehouse. The benchmark database consists of a fact table* `LINEORDER` *and four dimension tables, with foreign key constraints between the* `LINEORDER` *table and each of the dimension tables. There are 13 select-project-join-aggregate queries in the benchmark, split into 4 groups. For instance, Figure 2 shows the Query 4.3, which involves joins between all the tables in the database.*

Though structurally similar, different left-deep plans can have widely varying execution times, depending on the selectivities of the predicates in the query. For example, since the predicates on `CUSTOMER` *and* `DATE` *tables have selectivities* 20% *and* 28% *respectively, the plan in Figure 2a performs about* 6% *fewer hash table probes than that in Figure 2b.*

Each permutation of the dimension tables, called a *join order*, results in a distinct left-deep query plan. Given a permutation $\pi : \{1, 2, ..., n\} \rightarrow \{1, 2, ..., n\}$, we can use the notation $P_{\pi(1)\pi(2)...\pi(n)}$ to refer to the query plan corresponding to the join order $D_{\pi(1)}, D_{\pi(2)}, ..., D_{\pi(n)}$.

A key challenge faced by a query optimizer is picking the optimal join order among the $n!$ choices, which requires accurate estimation of predicate selectivities in spite of skew and correlation in the data distribution. This remains an open problem despite years of research [17, 23].

2.2 Modeling Performance Without LIP

We begin our analysis by modeling the impact of join order selection on the performance of query plans in the *naive evaluation strategy* that does not use the LIP technique. We focus on the star schema SPJ query shown in Equation 1.

Without loss of generality, we pick the plan $P_{12...n}$ as an arbitrary left-deep hash join tree for this query. In the theoretical results below, we have assumed that all data in the tables is independent and uniformly distributed. Under this assumption, the selectivity of the join between F and D_i is equal to the selectivity of the predicates on D_i. Our experiments in Section 5.3 show that this assumption does not affect the efficacy of LIP in handling real-world data with skew and correlation.

2.2.1 Cost Model

Let us use a simple model to derive the costs for the different operations in the execution of the query plan $P_{12...n}$. In this model, we assign unit cost each per tuple to the operations of checking whether it satisfies a selection predicate, inserting it into a hash table, and probing whether it is contained in the hash table.

As an example, consider a query with only one dimension table D containing 100 tuples. If 10 tuples pass the selection predicate and are inserted into a hash table, the total cost of these operations in our model is 100 (for selection) +10 (for insertion) = 110 units. If this hash table is probed using a fact table containing 1000 tuples, the probe cost is 1000 units and the total cost is 1110 units.

2.2.2 Hash Table Build Phase

In the naive evaluation strategy, we first apply the selection predicates on the dimension tables and then build hash tables on the results. Denoting the selectivity of the predicate on D_i by σ_i, the build cost is:

$$\text{BuildCost}(P_{12...n}) = \sum_{i=1}^{n}(1 + \sigma_i)|D_i| \tag{2}$$

Note that this cost is the same for all join orders.

2.2.3 Hash Table Probe Phase

Next, we probe each of the hash tables in the join order specified by the plan. Let us assume the selectivity of the join predicate between F and D_i is also σ_i. Then, the cost of probing the hash table on D_1 using F is $|F|$, and the result has a cardinality $\sigma_1|F|$. The subsequent probes have costs $\sigma_1|F|$, $\sigma_1\sigma_2|F|$ and so on. Letting $\sigma_0 = 1$, the hash table probe cost in our model can be written as:

$$\text{HashTableProbeCost}(P_{12...n}) = \sum_{i=1}^{n}\sigma_0\sigma_1...\sigma_{i-1}|F| \tag{3}$$

2.2.4 Bounds on Cost of Any Plan

The total cost $T(P_{12...n})$ of this plan is the sum of the build and the probe cost terms.

$$\begin{aligned} T(P_{12...n}) &= \text{BuildCost}(P_{12...n}) \\ &\quad + \text{HashTableProbeCost}(P_{12...n}) \\ &= \sum_{i=1}^{n}(1 + \sigma_i)|D_i| + \sum_{i=1}^{n}\sigma_0\sigma_1...\sigma_{i-1}|F| \end{aligned} \tag{4}$$

It is intuitively clear (and can be proved by induction on n) that the HashTableProbeCost above is minimized when the probes are done in ascending order of selectivities, and maximized when using descending order. These sorted join orders therefore also have the minimal (or maximal) total costs, since the BuildCost is independent of the join order. Let us denote the best (minimal cost) join order by the sequence $1', 2', ..., n'$ and the corresponding plan $P_{1'2'...n'}$ by P_b. Then, the worst (maximal cost) join order is the reverse sequence $n', ..., 2', 1'$, and we denote $P_{n'...2'1'}$ as P_w. Let us also use $\sigma_{\min} = \sigma_{1'}$ and $\sigma_{\max} = \sigma_{n'}$ to denote the minimum and maximum selectivities.

$$\sigma_{\min} = \sigma_{1'} \leq \sigma_{2'} \leq ... \leq \sigma_{n'} = \sigma_{\max} \tag{5}$$

Replacing each selectivity factor σ_i for $i > 0$ in the summation in Equation 3 with $\sigma_{\max}$ (or $\sigma_{\min}$) gives us an upper bound (respectively, lower bound) for the cost of any plan.

$$\frac{1 - \sigma_{\min}^n}{1 - \sigma_{\min}}|F| \leq \text{HashTableProbeCost}(P_{12...n}) \leq \frac{1 - \sigma_{\max}^n}{1 - \sigma_{\max}}|F| \tag{6}$$

2.2.5 Robustness: Cost Difference Between Plans

Recall that the BuildCost is the same for all join orders. Thus, the difference in total cost between any two join orders

Table 1: Summary of Notation used in Sections 2 and 3

Category	Notation	Meaning	Remarks
Star Schema	F	fact table	see expression in Equation 1
	$D_1, ..., D_N$	dimension tables	
	n	number of joins in the query plan	
Bloom Filter	r	bit array size: number of bits per inserted object	configured by query optimizer
	k	number of hash functions	
	ϵ	false positive rate	
Query Plans	$1, 2, ..., n$	join order $D_1, D_2..., D_n$	represents any arbitrary join order
	$1', 2', ..., n'$	optimal join order (its reverse is the worst join order)	see Section 2.2
	$P_{12...n}$	query plan in the naive evaluation strategy without LIP	
	P_b, P_w	best and worst naive query plans	
	$B_{12...n}$	query plan using LIP, but no adaptive reordering	see Section 4.1
	B_b, B_w	best and worst query plans with LIP, no adaptive reordering	
Cost Model	1	cost per tuple for predicate evaluation and hash table insertion or probe	see Section 2.2
	β	relative cost of Bloom filter insertion or probe	
	$\sigma_1, ..., \sigma_n$	selection predicate and join selectivities	
	$\sigma_{\min}, \sigma_{\max}$	minimum and maximum selectivities among σ_i	
	BuildCost($\cdot$)	cost of selection, and building hash table and bloom filter	
	HashTableProbeCost(.)	cost of probing hash tables in LIP strategies	
	T($\cdot$)	total cost of query plan	
	BloomProbeCost($\cdot$)	cost of probing Bloom filters	see Section 4.1

is just the difference between their HashTableProbeCosts. This cost difference between the best and the worst plans is:

$$T(P_w) - T(P_b) = \sum_{i=1}^{n-1} \left(\sigma_{n'}...\sigma_{(n-i+1)'} - \sigma_{1'}...\sigma_{i'}\right)|F| \quad (7)$$

In Section 2.3 below, we will formally define a notion of robustness that depends on this cost difference. Note that this equation has $n-1$ difference terms in the summation. We will now find a lower bound for this expression in terms of $\sigma_{\max} - \sigma_{\min}$. The i^{th} difference term above is the difference between the terms $\sigma_{n'}...\sigma_{(n-i+1)'}$ and $\sigma_{1'}...\sigma_{i'}$. These terms are respectively the products of the largest and smallest i selectivities in the sorted sequence $1', 2', ..., n'$.

To obtain an upper bound, we begin by replacing all the $i-1$ factors apart from the first factor $\sigma_{n'}$ in the larger term with the corresponding $i-1$ smaller factors from the other term. Then, we replace these $i-1$ factors by the smallest selectivity $\sigma_{1'}$.

$$\begin{aligned}\sigma_{n'}\sigma_{(n-1)'}...\sigma_{(n-i+1)'} - \sigma_{1'}\sigma_{2'}...\sigma_{i'} &\geq (\sigma_{n'} - \sigma_{1'})\sigma_{2'}...\sigma_{i'} \\ &\geq (\sigma_{n'} - \sigma_{1'})\sigma_{1'}^{i-1}\end{aligned}$$

Plugging the above bound into Equation 7, we get the following lower bound:

$$\begin{aligned}T(P_w) - T(P_b) &\geq \sum_{i=1}^{n-1} \sigma_{1'}^{i-1}(\sigma_{n'} - \sigma_{1'})|F| \\ &= \frac{1 - \sigma_{\min}^{n-1}}{1 - \sigma_{\min}}(\sigma_{\max} - \sigma_{\min})|F| \quad (8)\end{aligned}$$

2.3 Robustness

We will see that, in general, the difference in execution cost between the worst and the best plans grows linearly with the size of the fact table. Further, it also grows with the spread of selectivities of the predicates used in the query, i.e., $\sigma_{\max} - \sigma_{\min}$, because intuitively, errors in join order selection are more disastrous to performance when the selectivities are more different from each other. In fact, if we assume that the selectivities (and hence cardinalities) are estimated with some error tolerance δ, i.e., if each estimate $\hat{\sigma_i}$ is within δ of actual σ_i, then we expect the worst plan to be worse than the best in proportion to δ.

Our cost model does not incorporate the second order effects such as the size of the hash table on probe cost and the impact of number of output attributes on materialization cost. In the special case when the predicate selectivities are all similar, $\sigma_{\max} - \sigma_{\min}$ is negligibly small or even 0, and these second order effects dominate the cost difference between plans. Addressing this shortcoming of our model is left out of scope of this paper. Consequently, we assume that $\sigma_{\max} - \sigma_{\min}$ is non-zero in the definitions below.

To make this notion more concrete, let us formally define robustness such that we can use it to compare different query evaluation strategies.

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.*

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

For robustness, we want to pick an evaluation strategy which guarantees that mistakes made by the query optimizer are not too expensive. But a θ-fragile strategy nec-

essarily has a large spread in performance between the best and worst plans, particularly when θ is large. Thus, such a strategy adds fragility to plan selection. On the other hand, a Θ-robust strategy guarantees that even the worst plan in the plan space is not much more expensive than the optimal, particularly when Θ is small.

By Equation 8 above, the naive (non-LIP) evaluation strategy is θ-fragile with respect to the space of left-deep hash join trees, for $\theta = \frac{1-\sigma_{\min}^{n-1}}{1-\sigma_{\min}}$. In Section 4.1.5, we will show that the proposed LIP strategy is Θ-robust for a Θ typically smaller than this θ.

2.4 Bloom Filter

A Bloom filter [8] is a probabilistic data structure that succinctly represents a set. It is used to maintain approximate information about membership in the set. When probed to test for membership of an object in the set, if the Bloom filter returns `false`, then the object is guaranteed not to be a member of the set. It is also guaranteed to return `true` for all members of the set. But the filter may also wrongly return `true` for objects that are not members. The probability of such *false positives*, denoted ϵ, can be fixed by appropriately configuring the filter.

A Bloom filter consists of a configurable number of bits in a bit array, as well as a configurable number k of hash functions. These k hash functions can be thought of as mapping each object into k bit positions in the bit array. To insert an object into the filter, we set the k bits at the positions indicated by the k hash functions. To test whether an object is in the filter, we check whether *all* of the corresponding k bit positions are set. If any of these k bit positions is unset, then we can be sure that the object is not in the set. But it is possible that all these k bits are set even though the object had never been inserted, resulting in a *false positive*.

Theoretical results such as those in [20] can be used to find the optimally sized Bloom filter configuration for a given target false positive rate.

3. LOOKAHEAD INFORMATION PASSING

We briefly motivated and presented an overview of *Lookahead Information Passing* (LIP) in Section 1. In this section, we define this strategy more concretely.

The key insight behind the LIP strategy is that in the space of left-deep join trees for star schema queries, a suboptimal plan schedules less selective joins before selective ones. Such a plan incurs additional cost relative to the optimal one due to extra hash table probes for tuples that are filtered out in the later joins.

Thus, we can mitigate this cost by forwarding information about later join predicates to earlier ones in the plan. Such a *lookahead filter* can be forwarded from the build tables involved in downstream joins to the probe table, where they can be applied prior to performing the hash table probe. The resulting hash table probes now involve far fewer tuples.

3.1 LIP Algorithm Summary

We now give an overview of the LIP strategy.

1. **Build Phase.** For each dimension table in the join tree, we build both a hash table as well as a succinct filter data structure, such as a Bloom filter, on the selection result. This phase is discussed in detail in Section 3.1.1.
2. **Bloom Filter Probe Phase.** We then simultaneously probe all these filters using the fact table, maintaining hit/miss statistics.
3. **Adaptive Reordering Phase.** We adaptively reorder the filters during the probe, using the estimated selectivity. For a good choice of filter configuration, the result of this multiway filter probe is roughly equal to the final output result, albeit possibly with a few false positives. We describe this algorithm in more detail in Section 3.2.
4. **Hash Table Probe Phase.** Subsequently, we probe the hash tables and eliminate the false positives as well as collect build-side attributes that are required for further processing.

Thus, using succinct filter data structures (such as a Bloom filter), we can greatly reduce the hash table probe cost (which is the dominating cost term) in such multi-join queries. In fact, as we show in Section 4.2.2, there are even fewer hash table probes than in the optimal plan using naive evaluation. However, we bear the additional cost of building and probing the LIP filter itself. On balance, we still see a speedup since the LIP data structures (Bloom filters in this paper) are more space-efficient than hash tables, and are more likely to fit in the processor caches, minimizing probe costs.

The small size of such filters also allows us to dynamically reorder their probes based on their observed selectivities. This adaptive reordering ensures that the number of probes to the filters is close to the number of hash table probes required in the optimal join order, regardless of the join order picked by the optimizer. In fact, nearly all join orders exhibit roughly the same execution time, and that time is roughly equal to the optimal execution time (or better).

3.1.1 Critical Implementation Aspects

To efficiently parallelize the construction of Bloom filters in the Build Phase of the algorithm in a multithreaded execution environment, we use the commutativity and associativity properties of the Bloom filter insertion operation. Each thread, while scanning the input dimension table, constructs its own thread-local copy of the Bloom filter. All these local filters have the same configuration, set by the query optimizer. Finally, all the thread-local filters for a particular table are unioned together using the bitwise-OR operation on the bit array.

3.2 Adaptive Reordering of Lookahead Filters

3.2.1 Motivation

In the LIP strategy, we trade off expensive hash table probes for efficient lookahead filter lookups. However, these lookups are not negligibly cheap. Further, the number of lookups depends on the order in which the filters are probed, which can itself reduce robustness. For instance, if we always apply the lookahead filters in the same order as the join order, then a plan with a bad join order will cause a large number of tuples to be used for probing the low-selectivity filters, only to have them dropped in the following high-selectivity filter probes.

We now summarize the algorithm we use to mitigate this issue, followed by some implementation notes. Then, we use an analytical model to prove that the algorithm has fast convergence. Experimental results supporting the need for adaptiveness are presented in Section 5.4.

3.2.2 Algorithm Summary

Algorithm 1 shows how we dynamically adapt the lookahead filter probe order to mitigate the additional cost due to a bad fixed ordering. We maintain hit/miss statistics for all probes into each of the lookahead filters. Periodically, these statistics are used to estimate the observed selectivity of the underlying filters, called the *miss rate* here. Sorting the lookahead filters by their miss rates ensures that subsequent probes occur on the most selective filter first, then the next most selective filter, and so on. Since the convergence is usually very fast, the number of lookahead filter probes performed is therefore roughly the same as the number of hash table probes in the optimal (minimal cost) join order, regardless of the join order selected by the query optimizer. Thus, this adaptive reordering is a crucial contributor to the robustness and the near-optimality properties of LIP.

Algorithm 1: Filtering with adaptive reordering

Input: *filters* – an array of m lookahead filters
tuples – an array of n tuples
Output: indices of tuples that pass filtering

```
results ← ∅
foreach f in filters do
    count[f] ← 0
    miss[f] ← 0
batch_size ← 64
n ← |tuples|
loc ← 0
while loc < n do
    probe_batch ← an array of tuple indices from loc
                  to min(loc + batch_size, n) − 1
    foreach f in filters (in order) do
        result_batch ← ∅
        foreach i in probe_batch do
            if f contains tuples[i] then
                result_batch ← result_batch ∪ {i}
        count[f] ← count[f] + |probe_batch|
        miss[f] ← miss[f] + |probe_batch| - |result_batch|
        probe_batch ← result_batch
    Sort filters in ascending order of miss[f]/count[f]
    results ← results ∪ probe_batch
    loc ← loc + batch_size
    batch_size ← batch_size × 2
return results
```

3.2.3 Critical Implementation Aspects

We have implemented the LIP technique in the Quickstep RDBMS, whose storage subsystem horizontally partitions each table into small blocks of a few megabytes each. The algorithm above is run for each such block in the fact table. We begin the probe by creating a small batch *probe_batch* of a few hundred tuples. We then probe the first lookahead filter f using the batch of tuples, keeping statistics about the number of probes, and hits/misses in an auxiliary data structure. The tuples whose keys are found to be hits in f (including false positives) are written into a *result_batch*, which is then used to probe the next filter in bulk. After all the filters have been probed using the first batch, we sort the filters in ascending order of their miss rates, computed from the hit/miss statistics in the auxiliary data structures. This new ordering is used for the next cycle of batched probes.

The batching of tuples in the algorithm is necessary for efficiency because the aggregate size of all lookahead (Bloom) filters is often larger than the processor cache size. The probes are most cache-efficient when we allow one filter at a time to warm up the cache and become cache-resident by consecutively probing it with tuples in a batch. Note another implementation detail: to avoid the cost of copying tuples between *probe_batch* and *result_batch*, we only use a single *batch* data structure containing tuple references. Our execution engine performs the lookahead filter probes as part of the hash join probe operator for the bottom-most join in the query plan. After all the filters have been probed using a batch, the resulting tuples are used to probe the bottom-most hash table. The remaining hash tables are only probed after an entire output block is produced.

While large batch sizes benefit from cache residence of the Bloom filters, they slow down the convergence rate of the adaptive algorithm, since reordering is only done between batches. To balance the two effects, we adapt the batch sizes by iteratively doubling it at the end of every cycle, along with the adaptive reordering. Both the batch size and the lookahead filter order are reset after completing all the probes for a given storage block of the fact table.

3.2.4 Convergence Rates

To examine the convergence procedure in our model, we assume that join predicates are independent and that the tuples in a block are randomly distributed. The independence assumption ensures that the observed miss rate of a given filter does not depend on which filters were probed prior to it. The random distribution assumption ensures that, within a block, observed miss rates for the tuples at the beginning of a block are roughly the same as those for tuples anywhere else in the block. Note that this assumption does not require the tuples in the entire table to be randomly distributed: for instance, the adaptive algorithm can gracefully deal with partitioned tables or biases in the order of insertion of tuples into the fact table.

Under the above assumptions, according to the law of large numbers, the observed miss rate for the i^{th} filter converges to its selectivity, say γ_i. Note that in the case of a Bloom filter, this γ_i includes both the selectivity of the predicate on the corresponding dimension table, as well as a (configurable) false positive rate. Consider the probability that, after probing using N_i tuples, the observed miss rate $\hat{\gamma_i}$ is off from γ_i by more than a factor δ. Modeling the observed miss rate as a Binomial random variable with mean γ_i, using Chebyshev's inequality we can derive that:

$$\Pr\left(1-\delta < \frac{\hat{\gamma_i}}{\gamma_i} < 1+\delta\right) \geq \frac{1-\gamma_i}{N_i \gamma_i \delta^2}$$

We see that the observed miss rate for a filter converges to its true selectivity at a rate proportional of the number of tuples used to probe the filter.

Note that this number of probes for the i^{th} filter within a batch depends on the selectivities of the prior filters, in that highly selective prior filters $1, 2, ..., (i-1)$ may result in too few tuples being used to probe the i^{th} filter. However, in practice, for typical selectivities in the range 5% to 25%, we have found that the ordering of the filters converges to the

optimal in just 3-4 adapter cycles. For instance, if the true selectivity $\gamma = 0.10$, then after examining 3800 tuples, the estimation error δ is less than 5% with 95% probability.

4. MODELING PERFORMANCE OF LIP

In this section, we analyze the performance of query plans in the LIP strategy. This performance model not only allows us to compare this strategy with the naive one (analyzed in Section 2.2), but also to prove theoretical guarantees regarding robustness and near-optimality.

4.1 Robustness Through LIP

Corresponding to the plan $P_{12...n}$ in the naive evaluation strategy, consider the equivalent plan $B_{12...n}$ that uses LIP to whittle down the fact table before probing the hash tables in the order defined by the subscript sequence.

4.1.1 Cost Model

Let us model the per-tuple cost of a Bloom filter insertion or probe by a factor β relative to the unit cost for per-tuple hash table operations. In the remainder of this subsection, we separately analyze each phase of query evaluation in the LIP strategy, as highlighted in the steps of the algorithm summary in Section 3.1.

As an example, consider a query with only one dimension table D containing 100 tuples. Suppose 10 tuples pass the selection predicate and get inserted into the hash table and Bloom filter. The total cost of these operations in our model is 100 (for selection) + 10 (for hash table insertion) + 10 β (for Bloom filter insertion) = $110 + 10\beta$ units. If the Bloom filter is probed using a fact table containing 1000 tuples, the probe cost is 1000β units. Assuming a false positive rate of 10% for the filter, the probe results in 200 selected tuples, rather than the ideal 100 (extra 10% of 1000 tuples). These 200 tuples are then used to probe the hash table at a cost of 200 units. Thus, the total cost of this plan using the LIP technique is $310 + 1010\beta$ units.

4.1.2 Hash Table and Bloom Filter Build Phase

As a first step in evaluating a plan using the LIP strategy, we apply the selection predicates on the dimension tables and build both a hash table and a Bloom filter on each result. The total cost for these operations is independent of the selected join order and is given by:

$$\text{BuildCost}(B_{12...n}) = \sum_{i=1}^{n}(1 + \sigma_i + \beta\sigma_i)|D_i| \qquad (10)$$

4.1.3 Bloom Filter Probe and Reordering Phases

Before we analyze the next two phases of the algorithm, we make an observation about the cardinality of the result set of any Bloom filter probes. Recall from Section 2.4 that as a probabilistic data structure, the Bloom filter has a certain false positive rate, say ϵ, that we can appropriately configure. Consider the Bloom filter built on the selection result of the dimension table D_i. If we probe this filter using some N tuples from F, we would expect to obtain hits for not only the only at most N tuples can be hits, notwithstanding the false positive rate. Thus the selectivity of this Bloom filter is $\max(1, \sigma_i + \epsilon)$. For simplicity, we will assume that $\sigma_i + \epsilon < 1$ hereafter.

As proved in Section 3.2, the adaptive reordering algorithm converges quickly to the optimal ordering of Bloom filters. This ordering is the same as the increasing order of selectivities, which is also the optimal ordering we have seen before for hash table probes. As before, we denote this ordering by the sequence $1', 2', ..., n'$. We ignore the negligibly small overhead of the probes and adaptive algorithm until convergence. Thus the BloomProbeCost is:

$$\text{BloomProbeCost}(B_{12...n}) = \left[1 + \sum_{i=1}^{n-1}(\sigma_{1'} + \epsilon)...(\sigma_{i'} + \epsilon)\right]\beta|F| \qquad (11)$$

Note that this cost has the optimal selectivity order $1', 2', ..., n'$ on the right hand side, but is independent of the selected join order $1, 2, ..., n$. We note in passing that instead of probing the Bloom filters adaptively, if the join order were to be used for probing, then this cost term would no longer be independent of the join order. In fact, in that case, this cost term would actually dominate the overall query execution cost, greatly impacting robustness.

4.1.4 Hash Table Probe Phase

Regardless of the order of the Bloom filter probes above, the probes result in a final relation of size $(\sigma_1 + \epsilon)...(\sigma_n + \epsilon)|F|$, which we use, in the Hash Table Probe Phase, to probe the n hash tables built in the Build Phase. After probing the first hash table at a cost of $(\sigma_1 + \epsilon)(\sigma_2 + \epsilon)...(\sigma_n + \epsilon)|F|$ units, we eliminate the false positives obtained from the corresponding first Bloom filter, so that the resulting cardinality is $\sigma_1(\sigma_2 + \epsilon)...(\sigma_n + \epsilon)|F|$. Continuing this line of reasoning (and setting $\sigma_0 = 1$ for convenience), we get:

$$\text{HashTableProbeCost}(B_{12...n}) = \sum_{i=1}^{n}\sigma_0\sigma_1...\sigma_{i-1}(\sigma_i + \epsilon)...(\sigma_n + \epsilon)|F| \qquad (12)$$

If we ignore the terms that are $O(\epsilon^2)$ or higher powers of ϵ, we can simplify the above equation to:

$$\text{HashTableProbeCost}(B_{12...n}) \simeq \sigma_1\sigma_2...\sigma_n|F|\sum_{i=1}^{n}1 + \epsilon\left(\frac{1}{\sigma_i} + \frac{1}{\sigma_{i+1}} + ... + \frac{1}{\sigma_n}\right) \qquad (13)$$

This HashTableProbeCost is minimized when the probes are done in ascending order of selectivity $1', 2', ...n'$ and maximized when using the descending (reverse) order. Replacing each of the terms in the inner summation of the above equation with $\sigma_{n'} = \sigma_{\max}$ gives us a lower bound for the HashTableProbeCost of *any* plan with LIP.

$$\begin{aligned}
&\text{HashTableProbeCost}(B_{12...n}) \\
&\geq \sigma_1\sigma_2...\sigma_n|F|\sum_{i=1}^{n}1 + \epsilon\left(\frac{1}{\sigma_{n'}} + \frac{1}{\sigma_{n'}} + ... + \frac{1}{\sigma_{n'}}\right) \\
&= \sigma_1\sigma_2...\sigma_n|F|\sum_{i=1}^{n}1 + \epsilon\frac{n-i+1}{\sigma_{n'}} \\
&= \sigma_1\sigma_2...\sigma_n|F|\left[n + \frac{\epsilon}{\sigma_{\max}}\frac{n(n+1)}{2}\right] \qquad (14)
\end{aligned}$$

Substitution with $\sigma_{1'} = \sigma_{\min}$ in the inner terms gives us the following upper bound for the cost of *any* plan in this

strategy.

$$\text{HashTableProbeCost}(B_{12\ldots n}) \leq \sigma_1\sigma_2\ldots\sigma_n|F|\left[n + \frac{\epsilon}{\sigma_{\min}}\frac{n(n+1)}{2}\right] \quad (15)$$

4.1.5 Robustness: Cost Difference Between Plans

The total cost $T(B_{12\ldots n})$ for any plan is the sum of the three cost terms above. Since the first two terms are both independent of the join order, the difference between the total costs for any two plans is only due to the difference in the HashTableProbeCost terms. We can use the above upper bound for the best query plan B_b and the lower bound for the worst query plan B_w to bound the difference between the HashTableProbeCosts of any two plans in this strategy.

$$\text{T}(B_w) - \text{T}(B_b) \leq \frac{1}{2}\sigma_1\sigma_2\ldots\sigma_n\epsilon n(n+1)\left[\frac{1}{\sigma_{\min}} - \frac{1}{\sigma_{\max}}\right]|F| \quad (16)$$

Key Result: From Equation 16, it is clear that LIP with adaptive reordering is a Θ-robust evaluation strategy, for

$$\Theta = \frac{1}{2}\frac{\sigma_1\sigma_2\ldots\sigma_n}{\sigma_{\min}\sigma_{\max}}\epsilon n(n+1) \quad (17)$$

4.2 Insights from the Analytical Model

The simple cost model we have used allowed us to derive interesting analytical results for the performance of query plans using naive strategy in Section 2.2 and that for the LIP strategy in Section 4. Comparing these results, we now summarize the key insights obtained from this analysis.

4.2.1 LIP Makes Plans More Robust

Using Equation 8 in Section 2.3, we showed that the naive evaluation strategy without LIP is θ-fragile with respect to the space of left-deep hash join trees, for

$$\theta = \frac{1-\sigma_{\min}^{n-1}}{1-\sigma_{\min}}. \quad (18)$$

On the other hand, using Equation 16 above, we showed that LIP with adaptive reordering is Θ-robust with respect to the same plan space, for Θ given by Equation 17.

Usually, the selectivities σ_i are fairly small, lying in the range of 5% to 30%. In such a scenario, the product of $n-2$ selectivities in Θ bound for LIP strategy is likely to be much smaller than the factor quadratic in the number of joins n. Further, the false positive error rate ϵ can be made arbitrarily small by appropriately configuring the Bloom filter. In fact, it can even be made 0 by using an exact LIP data structure (such as a bitmap). To illustrate this point, let us take an example query with $n = 6$ joins with selectivities 5%, 10%, ..., 30%. Then, P_w and P_b have a cost difference of at least $0.21|F|$ units, whereas B_w and B_b have a cost difference of at most $0.002|F|$.

From this discussion, it is clear that *LIP theoretically guarantees robustness, whereas the naive evaluation strategy is likely to make plan selection much more fragile.*

4.2.2 LIP Makes Plans Nearly Optimal

The above discussion shows that the LIP technique dramatically improves the robustness of query plans by ensuring that all plans have nearly the same cost. We now show a stronger guarantee - that these costs are typically quite close to the cost of the optimal query plan using naive evaluation.

Using similar methods as above, we can get a lower bound for the total cost of the optimal naive query plan P_b, as well as an upper bound for the worst LIP query plan B_w.

$$T(P_b) \geq \sum_{i=1}^{n}(1+\sigma_i)|D_i| + \frac{1-\sigma_{\min}^n}{1-\sigma_{\min}}|F| \quad (19)$$

$$T(B_w) \leq \sum_{i=1}^{n}(1+\sigma_i+\beta\sigma_i)|D_i| + \frac{1-(\sigma_{\min}+\epsilon)^n}{1-(\sigma_{\min}+\epsilon)}\beta|F| + \sigma_1\sigma_2\ldots\sigma_n\left[n + \frac{\epsilon}{\sigma_{1'}}\frac{n(n+1)}{2}\right]|F| \quad (20)$$

In the above bounds, we see that the BuildCosts differ by $\sum_{i=1}^{n}\beta\sigma_i|D_i|$. Since the dimension tables are typically much smaller than the fact table, particularly after application of a selection predicate, this cost difference is usually a small fraction of the optimal cost.

As noted before, in the BloomProbeCost(B_w), the second term (ϵ) can be made very small, so that this term is roughly β times HashTableProbeCost(P_b). This is because as long as the false positive rate ϵ is small, we make roughly the same number of probes into the Bloom filter in any query plan using LIP with adaptive reordering as we make into the hash tables in the optimal naive query plan P_b. However, due to its small size, the Bloom filter is likely to be cache resident and hence make the probes much faster, i.e., $\beta \ll 1$. But, there is a tradeoff here as making ϵ smaller by increasing the Bloom filter size or number of hash functions also makes β larger (i.e., probes become more expensive).

Finally, as we have discussed before, the product of selectivities in the upper bound for HashTableProbeCost(B_w) is typically small enough to render the term a negligibly small fraction of the total cost of the optimal plan, despite the quadratic dependence on the number of joins n.

Summing the three terms, we see that $T(B_w)$ is typically at least as small as $T(P_b)$, and is often better. In other words, *not only is an LIP query plan guaranteed to run in approximately the time for the optimal LIP query plan, it is also nearly always as good as (and often better than) the optimal naive query plan.* Our empirical evaluation in Section 5 also confirms these theoretical insights.

5. EVALUATION

All the experiments presented in this section were performed using the Quickstep database engine, which is an in-memory relational DBMS. The experiments were run on a machine with 160 GB of main memory and dual socket Intel Xeon E5-2660 v3 processors, with 10 physical cores per socket (i.e. 40 virtual cores with hyperthreading).

We used three datasets for the experiments: a) Star Schema Benchmark (SSB) [22] at various scale factors, b) a synthetic dataset to stress different data parameters, and c) TPC-H at scale factor 100. In the description below, unless stated otherwise, we use the 100 scale factor SSB dataset.

In all the experiments, the buffer pool was large enough to contain the entire working set in memory. All reported query

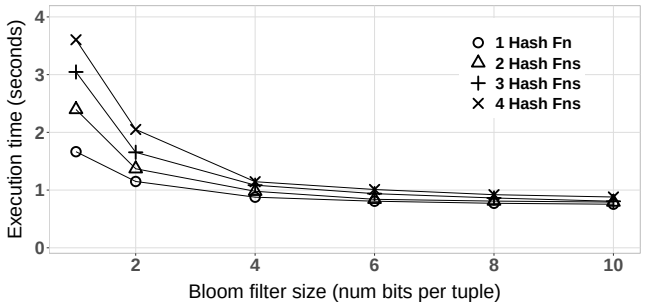

Figure 3: Comparison of optimal execution times for Query 4.3 using different Bloom filter configurations

execution times are averages of 5 successive runs. Since we are interested in how robust our techniques make the query execution times, we enhanced the query optimizer to produce all possible join orders for each query. All joins were performed using a hash join algorithm, where the fact table was used to probe the hash tables built on the dimension table(s). LIP was configured to use Bloom filters with 1 identity hash function and size 8 bits per tuple estimated in the dimension table (after selection). This configuration choice is explained in Section 5.1 below.

5.1 Choice of Bloom Filter Configuration

The configuration of a Bloom filter is defined by its size (r) as well as the number (k) and choice of hash functions. These parameter choices affect not only its false positive rate (denoted ϵ in this paper) but also the computational cost of operations on it (denoted β). While there are theoretical results [20] about the former, optimizing the latter requires empirical study. In this paper, we are interested in the overall execution cost for a query, which depends on both these factors.

Figure 3 shows the optimal execution time (i.e., for the best join order) among all 24 possible join orders for SSB Query 4.3, for different Bloom filter configurations. (Results for other queries are similar, and in the interest of space, we only present results for Query 4.3.) The four curves in the figure show the execution times when the number of hash functions used for building and probing the Bloom filter is varied from 1 through 4. The x-axis for the plots is the size of the Bloom filter, as a ratio of the estimated cardinality of the selection result for the dimension table it is based on, i.e., the number of bits in the Bloom filter per tuple used to build the corresponding hash table.

It is clear that the best performance is obtained using just a single hash function, and about 8-10 bits per tuple. Further, the performance of the query is not sensitive to the size of the Bloom filter near this optimal size. This latter observation justifies the use of our technique even when the estimated cardinality of the table may be erroneous.

Another factor that determines performance of the LIP approach is the choice of the hash functions. In our experiments with various different hash function families, we found that the identity function yields the best performance for this dataset. (The results in Figure 3 are for the first hash function being an identity function, and the others being variants of Knuth's multiplicative hash functions [16].)

The fact that the optimal Bloom filter configuration uses just a single hash function, and that too an identity function, may seem counter-intuitive. However, it is important to note that the Bloom filters are only used as an efficient but approximate filter to avoid more expensive hash table probes. This configuration choice minimizes the computational cost of evaluating the hash function and looking up the bit array, while sufficiently reducing the false positive rate to dramatically reduce the number of hash table probes.

5.2 Robustness to Join Order Selection

For each query in the SSB workload, we enumerated all possible join orders and ran them with and without LIP. Figure 5 shows the execution times for the first three queries (group 1) which have only one join. For the other queries, Figure 4 compares running times for all 24 join orders. In Figure 4, the execution times of query plans without using LIP is shown using diamonds, on the left in each subfigure, and they are connected to the execution times of query plans when using LIP (the latter is shown using circles).

In the naive strategy, we see that for all queries where there are multiple possible join orders (i.e., excepting the queries in group 1), there are several query plans that are far worse than the optimal one. For instance, in case of Query 4.1, the worst plan (18.1s) is more than 6× worse than the best plan (2.9s), and 10 of the 24 possible join orders have a running time at least double the optimal. On the other hand, the LIP strategy is much more robust to the join order. For the same query 4.1, all the 24 query plans have times within 5% of each other.

It is also remarkable that for most queries, the execution time of even the worst query plan using LIP is smaller than (or within the experimental error bounds) of the best query plan without LIP. In fact, comparing the total execution time for the entire benchmark, even choosing the worst query plan for every query along with LIP is better (17.4s) than choosing the optimal query plan for every query without LIP (17.6s), and far better than the worst plan for every query (90.9s).

5.3 Handling Skew and Correlation

In addition to the experiments using SSB, we also use a synthetic workload to stress-test our implementation using a large number of tables of widely-varying sizes, containing data with skew and correlation.

Database. The synthetic data generator creates a star schema database consisting of a fact table and n dimension tables (we set $n = 12$). Each dimension table has three columns, one integer primary key and two character columns with 25 distinct values each. They vary in size (4 each have sizes in ratio 1:10:100). The fact table has a primary key as well as all 12 foreign keys and is 10x larger than the largest dimension table. The total database size is configurable, and the table cardinalities are then derived according to the size ratios given above (we set the database size to 10 GB).

Data Distribution. Each value in the character columns consists of a prefix and a suffix. The prefix is always picked uniformly at random, whereas the suffix is picked uniformly in half the tables and, in the rest, using an inverse exponential distribution (as in [23]). Further, in half the dimension tables, the two character column values are correlated, sharing the same prefix with probability 0.80. In the rest of the tables, these columns are independent. The foreign keys in the fact table are independent and uniformly distributed.

The 12 dimension tables are obtained using all possible combinations of size, distribution and correlation above.

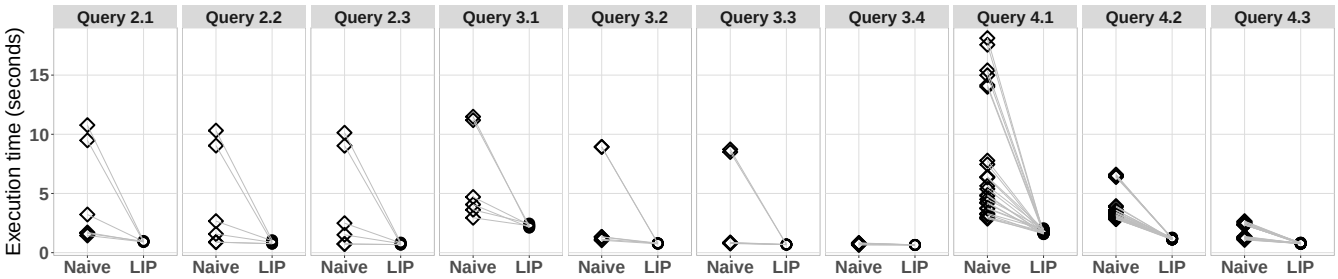

Figure 4: Execution times for the Star Schema Benchmark queries for groups 2–4, with and without LIP.

Thus, there are four small, medium and large dimension tables each. For each size, there are two tables with uniformly distributed values in the first character column, and two with skewed distribution. Finally, for each pair of size and distribution, there is one table with independent columns and another with correlated columns.

Queries. We also generated a synthetic workload over this dataset in which each query is similar in structure to the SSB queries: each query joins the fact table with a subset of the dimension tables. To generate each query, we first pick a random subset of dimension tables (six tables on average). For each table, we pick an equality selection predicate. The predicate may be on a single column (in which case, we compare with either the most or least frequent value in the skewed distribution) or on two columns (in which case, we compare with either the pair of most frequent or least frequent values in the correlated distribution). Thus, our workload generator allows us to individually control and examine the effects of skew and correlation, with both high and low selectivity predicates. For brevity, we only show the results for 10 randomly selected such queries here.

Query Plans. For each query, we randomly sampled 26 different join orders from the plan space. Each join order was executed with and without LIP.

Results. As shown in Figure 6, for every query, the best plan in the sample set had an execution time between 200 and 400 ms. Using LIP improved the best time for each query by about 13% on average. Only three of the queries were negatively impacted, with a 4% slowdown at worst. More importantly from the perspective of robustness, the average difference in execution times between the worst and best plans went down from 14 seconds without LIP to just 150 ms with LIP.

5.4 Importance of Adaptiveness

In this experiment, we ran each SSB query using all possible join orders, with the order of LIP filter probing either fixed or chosen adaptively using Algorithm 1.

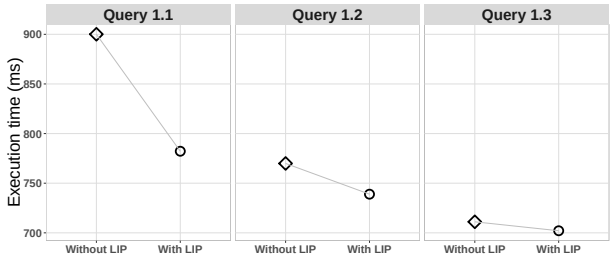

Figure 5: Execution times for SSB queries in group 1.

Figure 7 shows the resulting execution times for some selected queries. Overall, adaptive execution imposes little overhead (at most 8%). However, in some queries, such as queries 3.2 and 3.3, adaptiveness is crucial for maintaining robustness. For instance, for a particular join order for query 3.2, when we changed the bloom filter probes to be adaptive rather than the same fixed ordering as the join order, the slowdown from the optimal ordering dropped from 57.6% to just 6.3%. These results support our claim in Section 4.1 that adaptive reordering is a critical part of the robustness-enhancing property of the LIP technique.

5.5 Effect of Scaling

To better understand the scaling behavior of LIP, we ran the SSB workload for scale factors 10, 50, 100 and 200 (the largest scale factor possible on our machine), both with and without LIP. As shown in Figure 8, in both cases, we obtain better than linear scaling across this range of scale factors. In fact, the execution time for SF 200 was only 17× that for SF 10. Further, LIP not only improves the execution time across all scale factors, this effect actually scales up with the data size. For instance, going from SF 100 to SF 200 increases the speedup due to LIP from 34% to 63%.

5.6 Applying LIP to Subplans

Throughout this paper, we have limited our focus to left-deep join trees for star schemas. However, the LIP technique is more generally applicable to subplans with this pattern in larger query plans. To demonstrate this wider applicability, we picked Query 8 from the TPCH benchmark (scale factor 100) and applied LIP to star join subplans within each of the 675 query plans enumerated for this query by our optimizer.

Figure 9a shows one of the subplans following this pattern. While we have used `LINEITEM` as the "fact" table, the "dimension" tables in this pattern are themselves subplans having the same primary key as the `ORDERS`, `PART` and `SUPPLIER` tables. Note that while `ORDERS` is usually considered a fact table in the TPCH schema, for the purpose of LIP application, it can be considered as a dimension table due to the primary key - foreign key constraint between it and the `LINEITEM` table.

Figure 9b illustrates the gain in robustness using a box plot. With the naive execution strategy, the running times for this query varied from 2.1 s to 58 s, whereas LIP reduced this spread to between 1.3 s and 7.4 s. In addition to this 9× reduction in the difference between running times of plans, every query plan also saw an improvement in performance, with speedups varying from 1.20× to 18× (geometric mean of speedups was 4.0). In fact, 25% of the query plans ran faster with LIP than the optimal plan with naive evaluation.

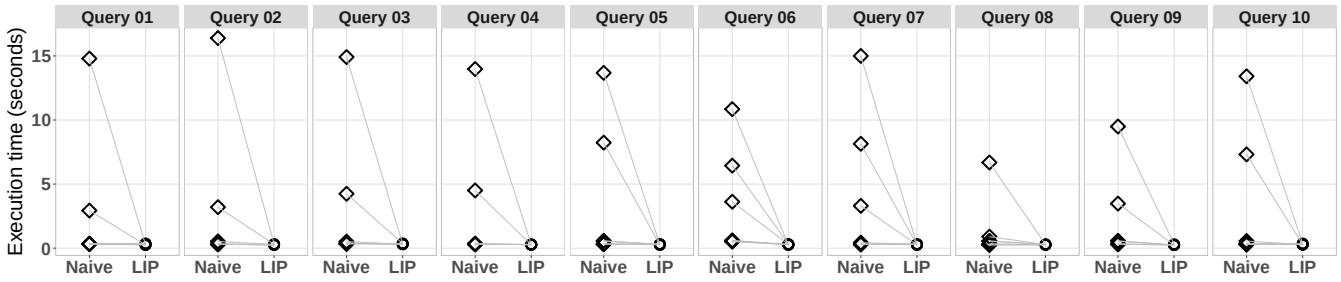

Figure 6: Execution times for sampled join orders of some queries in the synthetic workload.

5.7 Time Breakdown

To better understand the performance characteristics when using LIP, we broke down the fraction of overall query execution time spent in the build and probe phases for hash tables and LIP filters, as well as the time for materialization of intermediate and final results. Figure 10 shows the results of this experiment for all possible join orders for SSB Query 4.3. Similar results were obtained for other queries as well.

Across the board, we see a $5-10\times$ speedup from using LIP. In the naive strategy, the execution time is dominated by the cost of materializing results, accounting for 63% of the total execution time. Using LIP however, this fraction was down to just 4% on average, since almost all the redundant rows are eliminated early on. The cost of probing hash tables accounted for about 37% of the total time in the naive strategy but only a negligible fraction of the time in the LIP strategy. Instead, the execution time in LIP strategy was dominated by the cost of probing LIP filters (93% on average). It is worth noting, however, that this dominant cost term of probing LIP filters was only about 20% of the cost of probing hash tables in the naive strategy. Lastly, we observed that the cost of building hash tables and LIP filters was negligible in comparison to these other cost terms. These results validate our theoretical analysis and cost model: using LIP adds negligible overhead in terms of build cost, and more than pays off in terms of probe and materialization costs.

Finally, we note that both the hash table and LIP filter probe times showed little variance across join orders for the same query, highlighting the effect of the adaptive reordering algorithm.

6. RELATED WORK

We organize the related work in four groups, and discuss each group below.

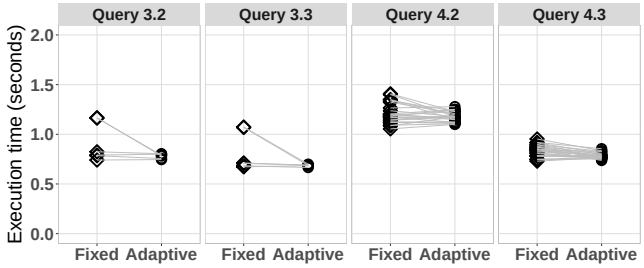

Figure 7: Execution times for fixed and adaptive lookahead filter probe ordering.

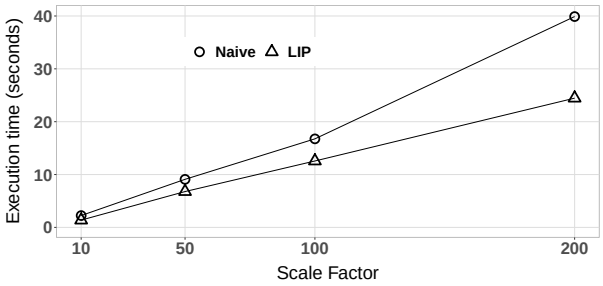

Figure 8: Execution times of the entire SSB workload for varyious scale factors, with and without LIP.

Bloom filters. First introduced in [8], Bloom filters have been used extensively in distributed database systems to minimize I/O and network transmission costs [18, 2, 9], as well as in semi-join optimization [7, 27, 18, 2, 9, 12, 25] to accelerate joins. We use bloom filters in LIP and also explore the impact of its parameters when used with LIP.

Sideways information passing (SIP). The *semi-join reduction* [7, 27, 18, 2, 9] accelerates a single join by passing a filter from one side to the other. SIP strategies and *magic sets transformation*, first introduced in [6], consider the space of semi-join reductions and associated query rewriting techniques. There has been much follow-on work in this space, including the use of greedy heuristics [10, 21] and cost-based approach [24].

Our proposed LIP strategy can be considered a special case of the general SIP strategies, though with the additional crucial component of adaptive filter reordering. Further, LIP also bears considerable likeness to *adaptive information passing* [14]. However, to the best of our knowledge ours is the first work to focus on the *robustness* benefits of SIP, as well as the ability to get *near-optimal* performance from all plans in the subspace under consideration. We supported these claims using both theoretical and empirical results.

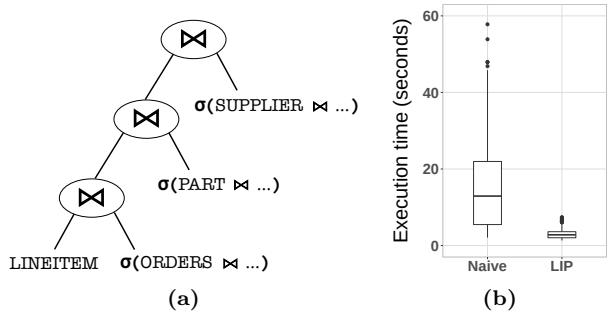

Figure 9: Left-deep star join subplan in TPCH Query 8, and execution times of all 675 possible query plans.

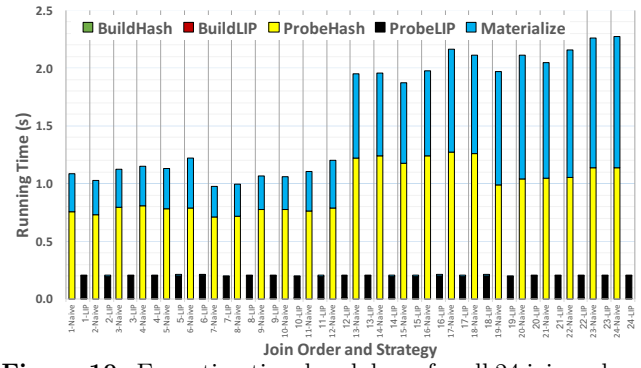

Figure 10: Execution time breakdown for all 24 join orders of SSB Query 4.3. For each query plan, the breakdown with the Naive and the LIP strategies are shown.

Adaptive reordering of filters. Prior work has shown the benefits from reordering of predicates in relational and stream processing. If selectivities are known exactly, then the optimal filter ordering can be derived directly [13]. Otherwise, learning-based approaches [1, 5] can be used. Our use of adaptive reordering of lookahead filters is inspired by these approaches. However, we focused our implementation efforts on ensuring low-overhead adaptation, as well as cache-sensitive bulk application of the filters. These implementation details are crucial to the fast convergence rate as well as the robustness benefits of LIP.

Robust query execution. The notion of robustness in query optimization has been well studied in the literature [28]. Proposed techniques include correcting cardinality estimates though sampling [3] or runtime feedback [26], dynamically switching between a candidate set of plans at runtime [11], as well as runtime re-optimizations [19, 4]. We consider our work to be complementary to such techniques. Whereas these techniques require a redesign of the query optimizer (in addition to any changes in the execution engine), our proposed LIP approach attempts to entirely avoid changes to the query optimizer, using query evaluation to immunize against selectivity estimation errors.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced a query execution strategy called LIP that collapses the space of left-deep query plans for star schema warehouses down to almost a single point near the optimal plan. In addition to this robustness benefit, it also significantly speeds up query execution in this important subplan space. We have demonstrated these claims through theoretical and empirical results. Besides the immediate application of LIP, we believe our work opens a novel approach to the notion of "robustness", one that is focused on query execution strategies possibly tailored to corresponding query plan (sub-)spaces. As part of future work, we hope to generalize these ideas to more complex schemas and query plans. We will also explore how this new approach to robustness impacts query optimization.

8. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant IIS-0963993. We thank the reviewers of this paper for their many insightful comments on earlier drafts of this paper.

9. REFERENCES

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD '00*, New York, NY, USA, 2000.
[2] E. Babb. Implementing a relational database by means of specialzed hardware. *ACM TODS*, 4(1), Mar. 1979.
[3] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD '05*, 2005.
[4] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD '05*, 2005.
[5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD '04*, 2004.
[6] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS '87*, New York, NY, USA, 1987.
[7] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1), Jan. 1981.
[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13, 1970.
[9] K. Bratbergsengen. Hashing methods and relational algebra operations. In *VLDB '84*, San Francisco, CA, USA, 1984.
[10] M.-S. Chen, H.-I. Hsiao, and P. S. Yu. On applying hash filters to improving the execution of multi-join queries. *The VLDB Journal*, 6(2), May 1997.
[11] A. Dutt and J. R. Haritsa. Plan bouquets: A fragrant approach to robust query processing. *ACM TODS*, 41(2), May 2016.
[12] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), June 1993.
[13] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD Rec.*, 22(2), June 1993.
[14] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE '08*, Washington, DC, USA, 2008. IEEE Computer Society.
[15] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.
[16] D. E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.
[17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *VLDB*, 9(3), Nov. 2015.
[18] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB '86*, San Francisco, CA, USA, 1986.
[19] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD '04*, 2004.
[20] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
[21] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD '09*, New York, NY, USA, 2009.
[22] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2009.
[23] T. Rabl, M. Poess, H.-A. Jacobsen, P. O'Neil, and E. O'Neil. Variations of the star schema benchmark to test the effects of data skew on query performance. In *ACM/SPEC International Conference on Performance Engineering*, 2013.
[24] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD '96*, New York, NY, USA, 1996.
[25] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the vertica analytic database: Lessons learned. In *ICDE*, 2013.
[26] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *VLDB '01*, San Francisco, CA, USA, 2001.
[27] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM TODS*, 9(1), Mar. 1984.
[28] S. Yin, A. Hameurlain, and F. Morvan. Robust query optimization methods with respect to estimation errors: A survey. *SIGMOD Rec.*, 44(3), Dec. 2015.