Accelerating Joins with Filters

Nicholas Corrado Xiating Ouyang

Abstract

In query optimization on star schemas, lookahead information passing (LIP) is a strategy exploiting the efficiency of probing succinct filters to eliminate practically all facts that do not appear in the final join results before performing the actual join. Assuming data independency across all columns in the fact table, LIP achieves efficient and robust query optimization. We present LIP-k, a variant of LIP that only remembers the hit/miss statistics for the previous k batches, achieving empirically efficient query execution on fact table with correlated and even adversarial data columns. We implemented LIP and LIP-k on a skeleton database on top of Apache Arrow and analyze the performance of each variant of LIP using the notion of competitive ratio in online algorithms.

1 Introduction

Performing join operations in database management systems is a fundamental and prevalent task in the modern data industry. Continuous efforts have been spent on building a reliable query optimizer over the last few decades. However, the current optimizers may still produce disastrously inefficient query plans which involves processing unnecessarily gigantic intermediate tables [**?**, **?**]. The *Lookahead Information Passing (LIP)* strategy aggressively uses Bloom Filters to filter the fact tables, effectively reducing the sizes of the intermediate tables. LIP is provably as efficient and robust as computing the join using the optimal query plan [**?**]. The key idea behind LIP is to estimate the filter selectivity of each dimension table and adaptively reorder the sequence of applying the filters to the fact table.

The filtering process can be modeled as the LIP problem in an online setting: Suppose we fix n filters, and the tuples in the fact table arrives in an online fashion. Upon arrival of each tuple, one has to decide a sequence of filters to probe the tuple, with an objective of minimizing the number of probes needed to decide whether to accept the tuple and forward it to the hash join phase, or to eliminate it. A mechanism deciding the sequence of applying the filters is thus crucial to the success of LIP. If a tuple passes all filters, *all* mechanisms must probe the tuple to all n filters to confirm its passage. If a tuple is eliminated by at least one filter, then the *optimal* mechanism would apply any filter that rejects the tuple first, using only one probe. Thus, given any fact table F, the number of probes that an optimal mechanism requires to process all tuples in the fact table can be readily computed:

$$\mathrm{OPT}(F) = n|F_{\mathrm{pass}}| + |F_{\mathrm{reject}}|,$$

where $|F_{\mathrm{pass}}|$ and $|F_{\mathrm{reject}}|$ are the number of tuples in F that pass all filters and are rejected in F respectively. For any mechanism $\mathcal{M}$, denoted by $\mathrm{ALG}_{\mathcal{M}}(F)$ the number of probes that $\mathcal{M}$ performed to process all tuples in the fact table. The performance of any mechanism $\mathcal{M}$ can thus be measured by

$$\max_{F} \frac{\mathrm{ALG}_{\mathcal{M}}(F)}{\mathrm{OPT}(F)},$$

called the *competitive ratio* of $\mathcal{M}$. The competitive ratio is always at least 1 by definition, and in this problem the competitive ratio is at most n, the number of filters, since one mechanism can probe each tuple to at most n filters.

This project aims at designing efficient LIP mechanisms and measure their performance in terms of their overall running time and competitive ratio. We will build a skeleton database system on top of Apache Arrow supporting LIP and hash-joins to conduct our experiments and test the performance of LIP-k—a variant mechanism of LIP that only remembers the statistics for the previous k batches—against the hash-join and the orignal LIP. We also present a theoretical result showing that no deterministic mechanism can have a competitive ratio better than n, and discuss possible extensions of LIP incorporating randomness to design a mechanism with better theoretical guarantee.

2 Lookahead Information Passing (LIP)

In this section, we first present the LIP strategy in [**?**]. We then discuss our variant LIP-k, designed to respond to local skewness more quickly than LIP. Finally, we discuss the competitive ratios of all deterministic mechanisms and provide proof on its lower bound. Possible extensions of LIP using randomness is also discussed.

2.1 LIP

The LIP strategy has three stages: (1) Build a hash table and a filter for each dimension table, (2) probe each fact tuple on the filters, producing a set of fact tuples with false positives, and (3) probe the hash table of each dimension table to eliminate the false positives. In what follows, we mainly discuss stage (2), since stages (1) and (3) are readily implemented by the filter constructors and database engine, respectively.

Let F be the fact table and D_i the dimension tables for $1 \leqslant i \leqslant n$. We denote the number of facts in F and each D_i as $|F|$ and $|D_i|$. Each LIP filter on D_i is a Bloom filter with false positive rate ε. The true selectivity σ_i of D_i on fact table F is given by $\sigma_i = |D_i \ltimes_{pk_i = fk_i} F|/|F|$, where pk_i is the primary key of D_i and fk_i is the foreign key of D_i in F. The `LIP-join` algorithm, depicted in Figure 1, computes the indices of tuples in F that pass all LIP filters. Note that because of the false positive rate ε associated with each filter, the set of indices is a superset of the true set of indices of tuples appearing in the final join result.

The partition in [**?**] satisfies that $|F_{t+1}| = 2|F_t|$ at line 5, and the algorithm approximates the true selectiveness σ_i of each dimension D_i using $\mathtt{pass}[i]/\mathtt{count}[i]$, the aggregated selectiveness since the beginning.

2.2 LIP-k

The LIP strategy in Figure 1 estimates the selectivity of each filter using statistics from all previous batches, which can be inefficient for certain foreign key distributions in the fact table. Consider the case where some filter f is very selective for the first T batches and not selective for the remaining batches. (For example, a filter f filtering for `year` $\geqslant 2017$ and the `Date` table is sorted in `year`.) In this case, LIP would obtain a good estimate of the selectivity of f during the first T iterations, and thus tend to apply f early in the remaining iterations. However, it is more efficient to postpone applying f in the remaining iterations, despite f has good selectivity in the first T iterations. One remedy to this is to only "remember" the hit/miss statistics of each filter over the previous k batches.

Empirical data shows that for the orignal SSB dataset and certain queries, LIP-k is as fast as LIP, and for certain datasets and queries LIP-k is faster than LIP. Detailed empirical data are presented and analyzed in Section 4.

PROCEDURE: LIP-join
INPUT: a fact table F and a set of n Bloom filters f_i for each D_i with $1 \leqslant i \leqslant n$
OUTPUT: Indices of tuples in F that pass the filtering

1. Initialize $I = \emptyset$
2. **foreach** filter f **do**
3. $\quad count[f] \leftarrow 0$
4. $\quad pass[f] \leftarrow 0$
5. Partition $F = \bigcup_{1 \leqslant t \leqslant T} F_t$.
6. **foreach** fact block F_t **do**
7. $\quad$ **foreach** filter f in order **do**
8. $\quad\quad$ **foreach** index $j \in F_t$ **do**
9. $\quad\quad\quad count[f] \leftarrow count[f] + 1$
10. $\quad\quad\quad$ **if** f contains $F_t[j]$
11. $\quad\quad\quad\quad I \leftarrow I \cup \{j\}$
12. $\quad\quad\quad\quad pass[f] \leftarrow pass[f] + 1$
13. $\quad$ **sort** filters f in nondesending order of $pass[f]/count[f]$
14. **return** I

Figure 1: The LIP algorithm for computing the joins.

2.3 Competitive Ratio Analysis

The LIP strategy and its variant LIP-k depicted in Figures 1 and 2 are *deterministic*, i.e. multiple executions over the same fact table would produce the same result. Experimental results show that LIP has faster execution time compared to hash join in the optimal sequence [**?**] on the benchmark dataset, in which the keys are distributed almost uniformly. However, it can be shown that any deterministic mechanism in the worst case can never achieve a competitive ratio less than n when played against an adversary producing an adversarial dataset.

Theorem 2.1. *Let n be the number of filters in the LIP problem. There is no deterministic mechanism $\mathcal{M}$ achieving a competitive ratio less than n for the* LIP *problem.*

Proof. We present an adversary to the arbitrary mechanism $\mathcal{M}$ such that $\mathcal{M}$ only achieves a competitive ratio of n in the worst case. Let the n filters be $f_1, f_2, \ldots, f_n$, and let S_k denote the filter sequence that will be used to filter batch k. Let each batch contain m tuples. At each iteration k, prior to LIP's probe phase, the adversary observes S_k and produces a batch of tuples $\{t_1, t_2, \ldots, t_m\}$, where $t_i \in f_n$ but $t_i \notin f_j$ for any $j < n$. The adversary then feeds this batch into mechanism $\mathcal{M}$. Thus, $\mathcal{M}$ performs n filter probes to eliminate each tuple, whereas the optimal sequence is to apply f_n first. Thus, $\mathcal{M}$ achieves a competitive ratio of n. □

It might be possible to design a randomized mechanism that can achieve a better competitive ratio than n. The randomized mechanism would, at the end of each batch, select a sequence of applying the filters from a distribution of all filter permutations, based on the estimated selectivities. However, we have not obtained any algorithmic upper bound on the competitive ratio. Furthermore, a randomized approach creates a new concern of being too computationally heavy.

PROCEDURE: LIP-k
INPUT: a fact table F and a set of n Bloom filters f_i for each D_i with $1 \leqslant i \leqslant n$
OUTPUT: Indices of tuples in F that pass the filtering

1. Initialize $I = \emptyset$
2. **foreach** filter f **do**
3. Initialize $count[f] \leftarrow 0$, $pass[f] \leftarrow 0$
4. Initialize $count\_queue[f]$ with k zeros and $pass\_queue[f]$ with k zeros.
5. Partition $F = \bigcup_{1 \leqslant t \leqslant T} F_t$.
6. **foreach** fact block F_t **do**
7. **foreach** filter f in order **do**
8. **foreach** index $j \in F_t$ **do**
9. $count[f] \leftarrow count[f] + 1$
10. **if** f contains $F_t[j]$
11. $I \leftarrow I \cup \{j\}$
12. $pass[f] \leftarrow pass[f] + 1$
13. $count\_queue[f].dequeue()$ and $pass\_queue[f].dequeue()$
14. $count\_queue[f].enqueue(count[f])$ and $pass\_queue[f].enqueue(pass[f])$
15. Reset $count[f] \leftarrow 0$, $pass[f] \leftarrow 0$
16. **sort** filters f in nondesending order of $sum(pass\_queue[f])/sum(count\_queue[f])$
17. **return** I

Figure 2: The LIP algorithm for computing the joins.

In the practical perspective however, one wishes to minimize the total running time of the mechanism, which is effectively the sum of the running time of the mechanism and the running time of building the filters and performing the probes. A trade-off between having a near optimal mechanism that consumes much time and allowing many failed probes to eliminate each non-participating tuple is therefore of much interest.

3 Database Implementation

We have developed a prototype database system supporting basic select and join operations on top of Apache Arrow [**?**], a column-store format. This minimal prototype is sufficient to benchmark the performance of our Hash join, LIP, and LIP-k.

Our implementations of LIP and LIP-k only support left-deep join tree plans where the fact table is the "outer table" in every join. We assume that the fact table schema contains foreign keys to all dimension tables, and each dimension table is single-key. Given a star schema fact table F and dimension tables D_i for $1 \leqslant i \leqslant n$, a join query in our system specifies selectors σ_F for F and σ_i for each D_i, and executing that query will return

$$\sigma_F(F) \bowtie \sigma_1(D_1) \bowtie \ldots \bowtie \sigma_n(D_n)$$

projected on the schema of F, *i.e.* we output the tuples in F that can be joined with each D_i. The supported

primitive selectors allow for selection ($=, \leqslant, \geqslant, <, >$) on scalar values and ranges. Range selections are executed using `BETWEEN` (ℓ, h), which selects all x with $\ell \leqslant x \leqslant h$.

The selectors for each dimension can be either a primitive selector consisting of simple predicate (*e.g.* `ORDER DATE` = 1997) or a composition (logical AND/OR) of multiple primitive/composite selectors. This is implemented using the Composite design pattern. Apache Arrow does not yet support vectorized string comparison operations nor vectorized range comparison operations. For queries involving string and/or range selections, we instead scan along the column, checking which rows satisfy the selection predicate. Such selections are inherently slower than the supported vectorized selections. Because all of our join implementations must implement row-wise selection for string and range predicates, all algorithms suffer the same slowdown. Thus, this implementation caveat does not preclude us from studying the relative performance of LIP and LIP-k.

We only support string and numeric data types. All numeric data is stored as 64-bit integers by default.

We do not fix the batch size but rather let Arrow determine the batch size. Arrow ddetermines the size of each individual batch at run-time, equal to the length of the smallest contiguous Arrow Array over all columns in the table. Thus, batch size is determined by (1) the size of the data in each column and (2) how Arrow chunks the underlying Arrays. Empircally, we find that the first 100 tend to contain 500 more tuples than later batches. The later batches contain roughly 10600 tuples.

The hash join algorithm first produces a hash table T_i for each $\sigma_i(D_i)$, projected on the k_i, and then probe each tuple in the fact table against all T_i. We used Sparseepp (accessible at https://github.com/greg7mdp/sparsepp) as our implementation of the hash table, in which the sparsehash by Google (accessible at https://github.com/sparsehash/sparsehash) is used as the underlying hash function. All primary keys are regarded as 64-bit integers.

For LIP and LIP-k, the succinct filter structure we choose is the Bloom filters. The default false-positive rate is set to 0.001, which requires 10 hash functions. We use Knuth's Multiplicative hash function, extended to accept a 64-bit integer as a seed. Our minimal implementation does not support selectivity estimation, so we initialize each Bloom filter assuming $\sigma(D_i) = \frac{1}{2}$, *i.e.* assuming half of the keys in each dimension table will be inserted into the filter.

Our code is available at https://github.com/NicholasCorrado/CS764.

4 Empirical Results

In this section we present the empirical results obtained from running Hash-join, LIP and LIP-k on several datasets. In Section 4.1, we present the datasets we use and describe how we generate skewed and adversarial datasets. In Section 4.2 we present the running time of multiple strategies and discuss their performance. In Section 4.3 we discuss how k affects the competitive ratio of LIP-k empirically on our datasets.

4.1 Datasets

The Star Schema Benchmark (SSB) is a variation of the TPC-H benchmark consisting of a large central fact table `LINEORDER` and four smaller dimension tables (`DATE`, `PART`, `CUSTOMER`, and `SUPPLIER`). The benchmark has 13 select-project-join queries split across four groups. We generate all tables in the benchmark using the SSB generator found at https://github.com/UWQuickstep/SQL-benchmark-data-generator/tree/master/ssbgen. We refer the `LINEORDER` table produced from the SBB generator with SF = 1 as `LINEORDER-UNIFORM`, since it's foreign keys for each tuple are sampled from a uniform distribution of possible keys (*i.e.* there is no skew).

Dataset Name	Skewed Foreign Keys	Description of Skew
LINEORDER-UNIFORM	N/A	No skew. Generated from SSB data generator with SF = 1.
LINEORDER-DATE-FIRST-HALF	ORDER DATE	$\sigma_{\text{SKEW PRED}}$ changes from 1 to 0 halfway through the table.
LINEORDER-DATE-50-50	ORDER DATE	$\sigma_{\text{SKEW PRED}}$ changes from 1 to 0 or from 0 to 1 every 50 batches.
LINEORDER-DATE-LINEAR	ORDER DATE	$\sigma_{\text{SKEW PRED}}$ increases linearly from 0 to 1 through the table.
LINEORDER-DATE-PART-ADVERSARY	ORDER DATE PART KEY	See Section 4.3

Table 1: Brief descriptions of all datasets studied.

We modify the LINEORDER-UNIFORM table to produce four additional skewed LINEORDER tables. For simplicity in generating skew tables, we skew the ORDER DATE foreign key in the origin LINEORDER table, corresponding to the primary key of the DATE table. We alter the distribution of ORDER DATE foreign keys satisfying DATE.YEAR = 1997 OR DATE.YEAR = 1998, which we shorten to SKEW PRED. The skewed tables are as follows:

- LINEORDER-DATE-FIRST-HALF: Let N be the total number of batches in LINEORDER. The first N/2 batches have $\sigma_{\text{SKEW PRED}} = 1$ while the remaining N/2 batches have $\sigma_{\text{SKEW PRED}} = 0$. In other words, The first N/2 batches contain only keys satisfying SKEW PRED, while the next N/2 batches contain only keys not satisying SKEW PRED, and so on.
- LINEORDER-DATE-50-50: The first 50 batches have $\sigma_{\text{SKEW PRED}} = 1$ while the next 50 batches have $\sigma_{\text{SKEW PRED}} = 0$, and so on.
- LINEORDER-DATE-LINEAR: $\sigma_{\text{SKEW PRED}}$ increases linearly from 0 to 1 across the LINEORDER table, *i.e.* batch k has $\sigma_{\text{SKEW PRED}} = k/N$.
- LINEORDER-DATE-PART-ADVERSARY: We defer discussion on this dataset to Section 4.3, as it requires a careful description of its construction.

All datasets are summarized in Table 1. All datasets use the same dimension tables originally produced from the SSB generator with SF = 1.

4.2 Execution Time

SSB query groups 1 and 2 do not join on ORDER DATE, and thus their execution times are unaffected by the skewed key column. Fig. 3 shows execution times for these queries on the LINEORDER-DATE-50-50 table. As expected, we observe that LIP and LIP-k have roughly equal performance for these queries. Henceforth, we will exclude query groups 1 and 2 from our analysis, as they provide us no insight into the relative performance of LIP and LIP-k. Nevertheless, from these results, we see that the extra overhead associated with LIP-k is negligible.

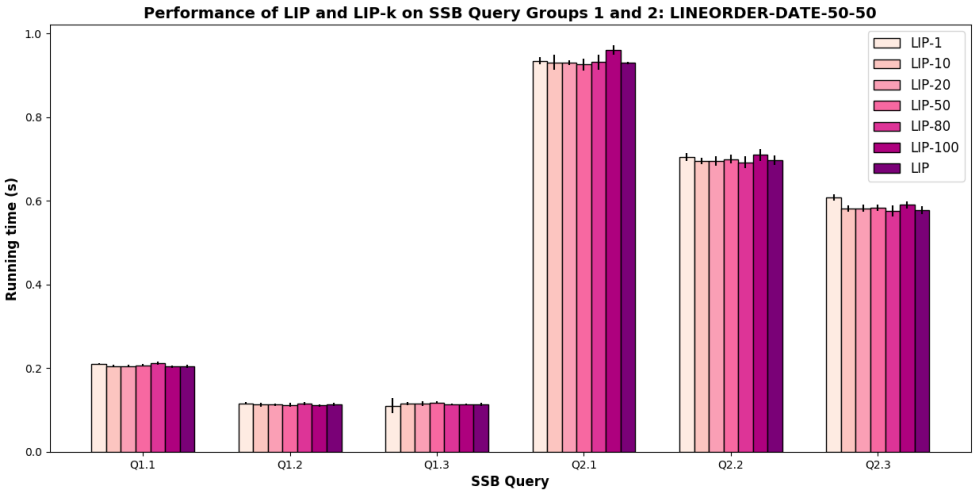


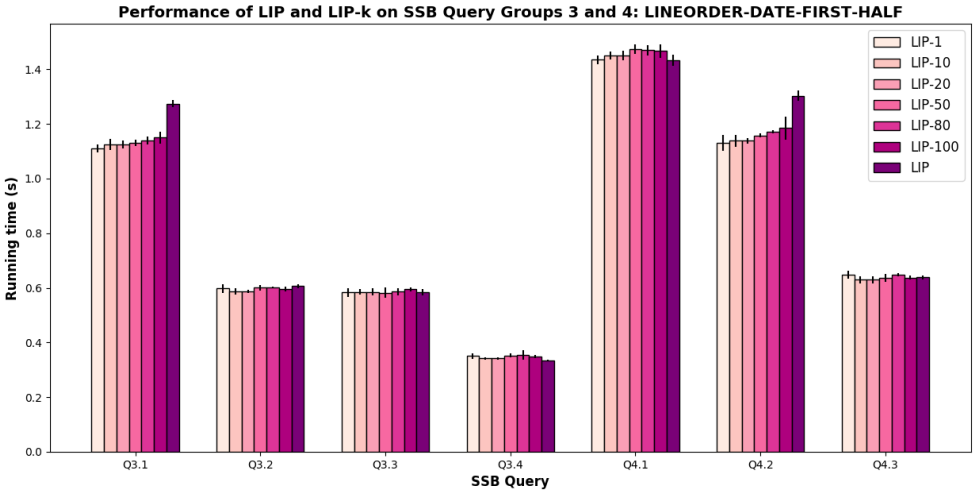
Figure 3: Execution time for SSB query groups 1 and 2 on `LINEORDER-DATE-50-50`

Query groups 3 and 4 — except for query 4.1 — join on `ORDER DATE`, so our skew can potentially affect the execution of LIP and LIP-k on these queries. Thus, we include only queries from groups 3 and 4 in our analysis. Query 4.1 is included as a "sanity check", since LIP and LIP-k should have roughly equal performance on this query.

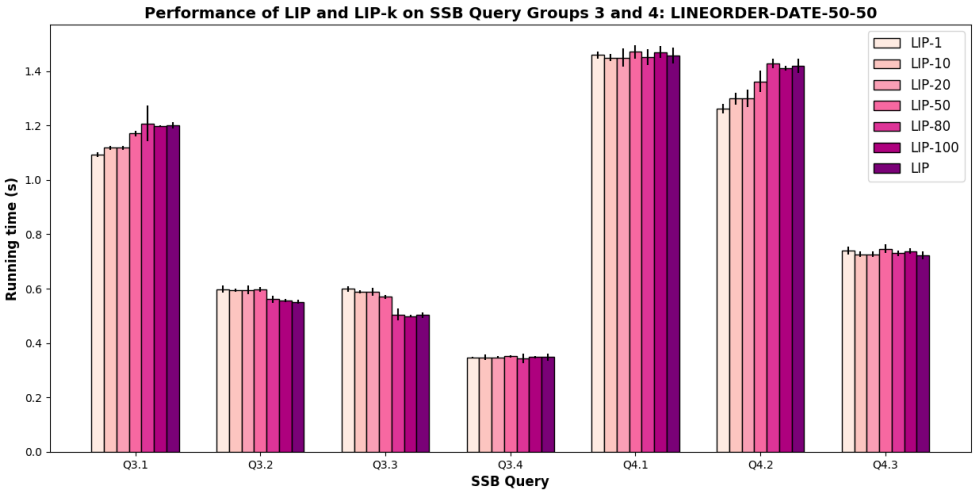
Fig. 4(a) shows execution times using `LINEORDER-DATE-FIRST-HALF`. We see that LIP and LIP-k have roughly equal performance on queries 3.2, 3.3, 3.4, and 4.3 (in addition to 4.1). In such queries, we do not gain much by responding to local changes in the key distribution, since there exists another filter, *e.g.* the `SUPPLIER` filter, that is very selective. LIP generally will not notice when $\sigma_i^{\texttt{DATE}} = 0$ and instead applies the `SUPPLIER` filter first. We observe that it is "good enough" to apply a very selective filter first, even if it is not optimal. On the other hand, LIP-k performs better on queries 3.1 and 4.2, since the other filters in these queries are not particularly selective. We also observe that smaller k has slightly better performance, because small k can respond much more quickly to the sudden change in selectivity.

Fig. 4(b) shows execution times using `LINEORDER-DATE-50-50`. LIP-k performs better than LIP on queries 3.1 and 4.2 for the same reasons state in the preceding paragraph. However, LIP performs better than LIP-k on queries 3.2 and 3.3. The key observation is that LIP-k must first "miss" before it can respond to a sudden change in selectivity. When the batch selectivity for the `DATE` column switches from 0 to 1, the `DATE` filter appears first in the filter sequence, and thus LIP-k performs an entire batch of unnecessary probes before recognizing that the `DATE` filter should not be probed first. We see that there is an inherent cost associated with responsiveness that may outweigh its benefit.

Fig. 5(a) shows execution times using `LINEORDER-DATE-LINEAR`. LIP and LIP-k have roughly the same performance for most queries, suggesting that both algorithms can adequately respond to smooth changes in key distributions. LIP-1 performs worse than all other algorithms on queries 3.3 and 4.3. because it does not compute an accurate selectivity estimate using only one previous batch.

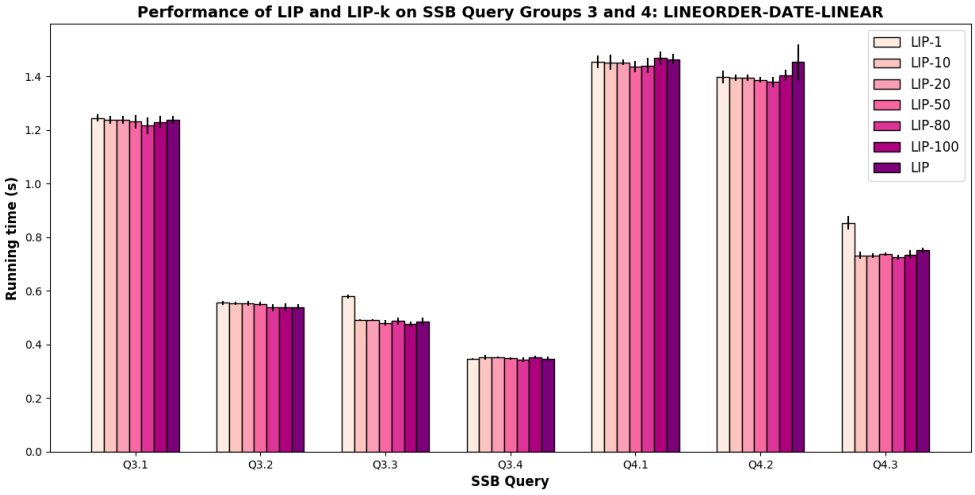


(a)

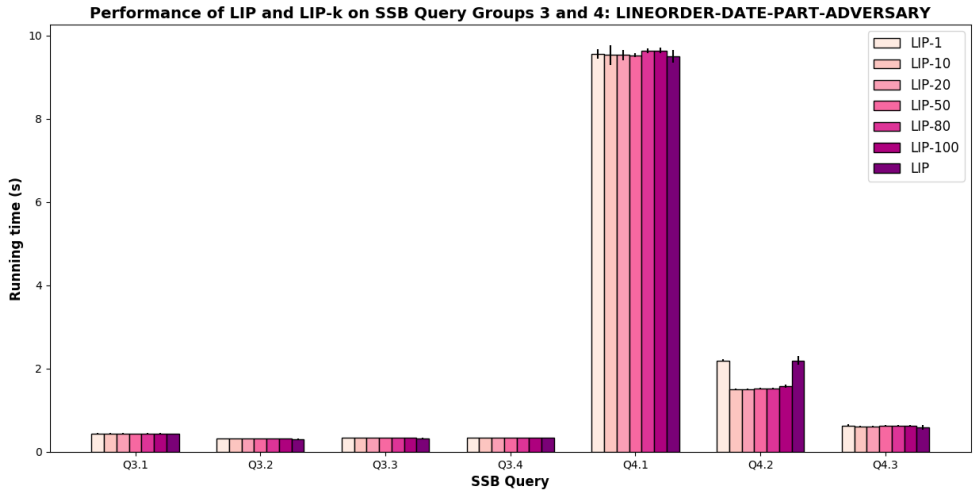


(b)

Figure 4: Execution time for SSB query groups 3 and 4 on (a) `LINEORDER-DATE-FIRST-HALF` and (b) `LINEORDER-DATE-50-50`



(a)



(b)

Figure 5: Execution time for SSB query groups 3 and 4 on (a) `LINEORDER-DATE-LINEAR` and (b) `LINEORDER-DATE-PART-ADVERSARY`

4.3 Competitive Ratio

To empirically support 2.1, we construct an adversarial dataset which forces LIP to perform the maximum number of filter probes possible. We now show how to generally construct such a dataset where two dimension tables are joined with the fact table ($n = 2$), mimicking the proof of Theorem 2.1.

Suppose we are joining our fact table with two dimension tables A and B using LIP. Let f_i^A denote the selectivity of the Bloom filter for A after processing the i^{th} batch. Let σ_i^A denote the selectivity of the Bloom filter for A on the i^{th} batch alone. Define analogous quantities for dimension table B.

We start with the first fact table batch having $\sigma_1^A = \frac{1}{2} - \varepsilon$ and $\sigma_1^B = \frac{1}{2} + \varepsilon$ where $0 < \varepsilon < \frac{1}{2}$. Then for all $j > 1$, we let

$$\sigma_j^A = \begin{cases} 1 & \text{if } j \text{ is even} \\ 0 & \text{if } j \text{ is odd} \end{cases} \quad \text{and } \sigma_j^B = \begin{cases} 0 & \text{if } j \text{ is even} \\ 1 & \text{if } j \text{ is odd} \end{cases}$$

Thus, the optimal filter sequence S^{OPT} for $j > 1$ is

$$S^{OPT} = \begin{cases} (B, A) & \text{if } j \text{ is even} \\ (A, B) & \text{if } j \text{ is odd} \end{cases}$$

After processing batch $j > 1$, we have $f_j^A = \frac{\frac{1}{2} - \varepsilon + \lfloor \frac{j}{2} \rfloor}{j}$ and $f_j^B = \frac{\frac{1}{2} + \varepsilon + \lfloor \frac{j}{2} \rfloor}{j}$ which can be rewritten as

$$f_j^A = \begin{cases} \frac{1}{2} + \frac{\frac{1}{2} - \varepsilon}{j} & \text{if } j \text{ is even} \\ \frac{1}{2} - \frac{\varepsilon}{j} & \text{if } j \text{ is odd} \end{cases} \quad \text{and } f_j^B = \begin{cases} \frac{1}{2} - \frac{\frac{1}{2} - \varepsilon}{j} & \text{if } j \text{ is even} \\ \frac{1}{2} + \frac{\varepsilon}{j} & \text{if } j \text{ is odd} \end{cases}$$

Since $\frac{1}{2} - \varepsilon > 0$, LIP's filter ordering for $j > 1$ will be

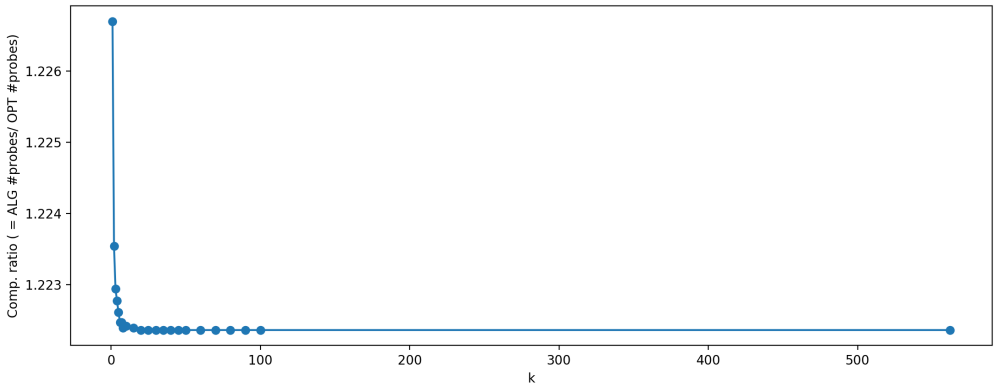
$$S = \begin{cases} (A, B) & \text{if } j \text{ is even} \\ (B, A) & \text{if } j \text{ is odd} \end{cases}$$

which is the reverse of S^{OPT}. Hence, after the first batch has been processed, LIP will have worst-case performance on all remaining batches. As the number of batches in the fact table increases, the competitve ratio of LIP on such a dataset approaches $n = 2$.

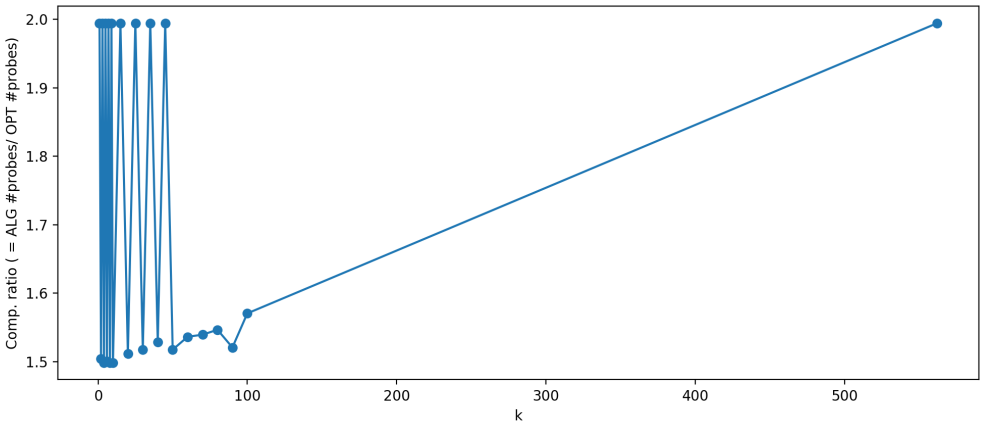
Following this construction, we generated an adversarial dataset for SSB query 4.2 with A = `DATE` and B = `PART`, excluding the joins on the `CUSTOMER` and `SUPPLIER` dimension tables. We exclude these two joins because it is much simpler to generate an adversarial dataset on fewer dimension tables. [1]

We ran LIP and LIP-k on the uniform and adversarial datasets and computed the competitive ratio of each algorithm. The results are depicted in Figure 6.

[1] In our implementation, we do not explicitly exclude the joins on `CUSTOMER` and `SUPPLIER` in query 4.2. Rather, we generate the adversarial dataset such that $\sigma_i^{\texttt{CUSTOMER}} = 1$ and $\sigma_i^{\texttt{SUPPLIER}} = 1$ for every batch i. This achieves the same effect as excluding the joins.



(a)



(b)

Figure 6: The competitive ratios of LIP-k against different k values. We ran LIP-k on uniform and adversarial datasets to produce (a) and (b) respectively. The data point at $k = 562$ represents LIP (which is essentially LIP-∞).

When the keys in the fact table columns are distributed uniformly, the filters need not react to the local changes. LIP-k with higher $k$produces a more accurate estimate of the selectivities than the LIP-k with smaller k. Hence, the competitive ratio decreases slightly as k increases, as depicted in Figure 6.

Figure 6(b) displays how an adversarial dataset can make LIP-k and LIP perform poorly. First, observe that LIP (*i.e.* LIP-562) achieves a competitive ratio of nearly 2. Our modified query 4.2 has two joins, and thus the performance of LIP matches the worst case competitive ratio.

LIP-k with odd k also achieves a competitive ratio of almost 2. For odd k, LIP-k's selectivity estimates always contains one more odd (or even) batch than the other, and thus the estimated selectivity and filtering sequence are in favor of the majority batch type, which produces the worst-case filter sequence on the following batch.

For even k, LIP-k's selectivity estimates contain an equal amount of even an odd batches, and thus estimated selectivities remain the same ($1/2$ by construction) throughout the execution, and the filter sequence does not change. Thus for at most half of the batches it is optimal, and for the other half it is worse, resulting in a competitive ratio of at least

$$\frac{1 \times 1/2 + 2 \times 1/2}{1} = 1.5,$$

as depicted in Figure 6.

When $i \leqslant k$, LIP and LIP-k execute identically, since LIP-k has not yet "forgotten" any previous batches. Thus, when $i < k$, LIP-k achieves a competitive ratio of 2 on each batch, regardless of wheter k is even or odd. When $i > k$ and k is odd, LIP-k still achieves a competitive ratio of 2 on each batch. However, When $i > k$ and k is even, LIP-k forgets to first batch and achieves a competive ratio of 1.5 on the remaining batches. This explains why the competitive ratio increases for even k as k increases. [2]

5 Future Works

A huge take-away from this project is that the budget allowed for filtering each filter is very limited, in the magnitude of near 100 CPU cycles. Hence maintaining the statistics that require heavy computation or using randomness that requires much computation cost are not practically beneficial. Given this, we will investigate dynamically start sampling a segment of the fact table to have true estimates of each filters, and then stick to this estimate for the next few batches. This has low maintenance overhead, and once the adaptive policy for dynamical sampling is established, we expect that to have better practical performance than LIP and LIP-k. Note that this strategy is still deterministic, still prone to the worst case n competitive ratio bound established by Theorem 2.1.

6 Concluding Remarks

The contributions of this project come in two-folds: We implemented Hash-join and LIP on top of Apache Arrow and thus provided an interface for future integration of Hustle on Apache Arrow; and we proposed LIP-k, a variant of LIP, which opens up an area for improving the performance LIP. It would also be interesting to study the problem of LIP in the online algorithmic setting to design an efficient mechanism utilizing randomness with better worst case guarantees.

[2] An astute observer might notice that the competitive ratio decreases for $k = 50$ and $k = 90$. This is because the batch size is not constant through execution.

Acknowledgements

The authors wish to thank Prof. Jignesh Patel for constant feedbacks on this project and Kevin Gaffney for helping us with Apache Arrow specifics. The second author wishes to thank Prof. Paris Koutris for the suggestion of working on a practical project when the lemma production pipe is jammed. It works.