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:

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

where $|F_{\text{pass}}|$ and $|F_{\text{reject}}|$ are the number of tuples in F that pass all filters and are rejected in F respectively. For any mechanism $\mathcal{M}$, denoted by $\text{ALG}_{\mathcal{M}}(F)$ 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

$$\max_F \frac{\text{ALG}_{\mathcal{M}}(F)}{\text{OPT}(F)},$$

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 $\sigma_i = |D_i \bowtie_{pk_i = fk_i} F|/|F|$, 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 **??**.

```
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.      $\mathrm{count}[f] \leftarrow 0$
4.      $\mathrm{pass}[f] \leftarrow 0$
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.              $\mathrm{count}[f] \leftarrow \mathrm{count}[f] + 1$
10.             if $f$ contains $F_t[j]$
11.                 $I \leftarrow I \cup \{j\}$
12.                 $\mathrm{pass}[f] \leftarrow \mathrm{pass}[f] + 1$
13.     sort filters $f$ in nondesending order of $\mathrm{pass}[f]/\mathrm{count}[f]$
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. $\square$

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