Accelerating Joins with Filters: Keeping a Limited Memory

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 using Star Schemas 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 [1, 4]. The *lookahead Information Passing (LIP)* strategy aggressively uses Bloom Filters to filter the fact tables to effectively reduce the sizes of the intermediate tables, provably as efficient and robust as computing the join using the optimal query plan [5]. 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 have to probe the tuple to all filters to confirm its passage; and if a tuple is eliminated by the filters, the *optimal* mechanism would apply any filter that rejects the tuple in the first place, 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.

In the practical perspective however, one wishes to minimize the total running time of LIP, 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.

This project aims at designing efficient LIP mechanisms and measure their performance in terms of their competitive ratio and their overall running time. 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 our variant LIP mechanisms against the hash-join and the orignal LIP.

2 Lookahead Information Passing (LIP)

The LIP strategy has three stages: (1) Building 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 either the database engine or the filter constructors.

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 $|D_i|$. A LIP filter on D_i is implemented using 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 the filtering of each Bloom filter of D_i. Note that there is an innate false positive rate ε associated with each Bloom Filter, and thus the set of indices is a superset of the true set of indices of tuples appearing in the final join result.

The partition in [5] 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 $pass[i]/count[i]$, the aggregated selectiveness since the beginning.

The LIP strategy is *deterministic* in Figure 1, i.e. multiple executions of LIP over the same fact table should all produce the same result. Experimental results show almost optimal performance compared to the performance of hash join in the optimal sequence [5]. However, it can be shown that deterministic mechanism in the worst case can never achieve a competitive ratio less than n.

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 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 consider n tuples $t_1, t_2, \ldots, t_n$, where $t_i \notin f_i$ but $t_i \in f_j$ for any $i \neq j$.

The adversary proceeds as follows: It first observes the sequence of filters σ_t at any step t set by the mechanism $\mathcal{M}$, and produce the input $f_{\sigma_t(n)}$ to the mechanism $\mathcal{M}$. Thus the mechanism $\mathcal{M}$ would require n filter probes to eliminate $f_{\sigma_t(n)}$ at each step t, whereas the optimal sequence is to apply $\sigma_t(n)$ at the first place. Thus it yields a competitive ratio of n. $\square$

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

Figure 1: The LIP algorithm for computing the joins.

3 LIP-k

LIP strategy in Figure 1 estimates the selectivity of each filter using statistics from the very beginning, which is inefficient for certain distribution and physical layout of data. Consider some filter f that is very selective for the first t_0 iterations at line 6 and not selective for the remaining iterations. (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_0 iterations, and thus tend to apply f early in the remaining iterations. However, it is in fact more efficient to postpone applying f in the remaining iterations, despite f has good selectivity in the first t_0 iterations. One remedy to this is to only "remember" the hit/miss statistics of each filter over the previous k iterations.

We remark that LIP-k is also deterministic, therefore prone to the worst case competitive ratio of n. However, 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 5.

4 Database Implementation

We have developed a prototype database system supporting basic join/select operations on star schemas sufficient to benchmark the performance of LIP and its variants on top of Apache Arrow. We assume that the fact table schema contains foreignkeys 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.

The supported premitive selectors are comparison with integer/string values ($=, \leqslant, \geqslant, <, >$) and between, where

```
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 $\mathrm{count}[f] \leftarrow 0$, $\mathrm{pass}[f] \leftarrow 0$
4.      Initialize $\mathrm{count\_queue}[f]$ with $k$ zeros and $\mathrm{pass\_queue}[f]$ 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.              $\mathrm{count}[f] \leftarrow \mathrm{count}[f] + 1$
10.             if $f$ contains $F_t[j]$
11.                 $I \leftarrow I \cup \{j\}$
12.                 $\mathrm{pass}[f] \leftarrow \mathrm{pass}[f] + 1$
13.         Dequeue one element from both $\mathrm{count\_queue}[f]$ and $\mathrm{pass\_queue}[f]$
14.         Enqueue $\mathrm{count}[f]$ and $\mathrm{pass}[f]$ to $\mathrm{count\_queue}[f]$ and $\mathrm{pass\_queue}[f]$ respectively
15.         Reset $\mathrm{count}[f] \leftarrow 0$, $\mathrm{pass}[f] \leftarrow 0$
16.     sort filters $f$ in nondesending order of $\mathrm{sum}(\mathrm{pass\_queue}[f])/\mathrm{sum}(\mathrm{count\_queue}[f])$
17. return $I$
```

Figure 2: The LIP algorithm for computing the joins.

the semantic of `BETWEE` (ℓ, h) is to select all x with $\ell \leqslant x \leqslant h$. The selectors for each dimension can be either a premitive selector, or a composition (logical AND and OR) of multiple primitive/composite selectors. This is implemented using the Composite design pattern.

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.

The succinct filter structure we choose is the Bloom filters. The default false-positive rate is set to 0.001 and the default number of inserts to the filter is set to 50,000. The hash function for the Bloom Filter is Knuth's Multiplicative hash function, extended to accept a 64-bit integer as a seed.

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

5 Empirical Results

The dataset for testing is obtained from the Star Schema Benchmark [3]. We will hard-code each queries in [3] to measure the join processing time.

6 Deploying lookahead filters in distributed systems

In this section we discuss a method to deploy the LIP filters in distributed systems. This method incorporates LIP and the HYPERCUBE algorithm [5].

Let p be the number of machines available. Let k_i be the primary key of D_i, and each tuple in the fact table F possesses a foreign key to each D_i. Suppose $p = \prod_{1 \leqslant i \leqslant n} p_i$, and then we label each of the p machines with a coordinate $(x_1, x_2, \dots, x_n)$ where each $1 \leqslant x_i \leqslant p_i$.

We first pick n hash functions h_i such that the range of each h_i is $\{1, 2, \dots, p_i\}$. Then for each dimension table D_i and for each primary key k_i in D_i, we send k_i to all machines with i-th component being $h_i(k_i)$. Hence

7 Concluding remarks

In this work, all variants have an adversarial fact table that would force the algorithm to achieve a competitive ratio of n. However, reasonable applications of some existing random online algorithm mechanisms may yield a competitive ratio of $O(\log n)$ using the weighted majority algorithm [2]. However, multiplicative updates on the weights may change the weighted average filter to use, but will not change the sequence of all filters if only comparing them based on their weights. More work is required here to either provide a mechanism to achieve a better competitive ratio, or provide a lower bound reduction showing that the competitive ratio of n is tight even for random mechanisms.

Acknowledgement

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 production of lemmas is jammed. It works.

References

[1] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[2] Nick Littlestone and Manfred K Warmuth. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

[3] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[4] Tilmann Rabl, Meikel Poess, Hans-Arno Jacobsen, Patrick O'Neil, and Elizabeth O'Neil. Variations of the star schema benchmark to test the effects of data skew on query performance. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 361–372. ACM, 2013.

[5] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M Patel. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *Proceedings of the VLDB Endowment*, 10(8):889–900, 2017.