1 Introduction

Join is one of the most important operations in the relational database systems, where many queries and analytical workloads rely on join operations upon two or more relations. However, joins are expensive, especially on large data with multiple relations. To address this high cost issue, Chaudhuri et al. [1999] introduced a promising approach which just return a simple random samples of the full results. However, there is a fundamental barrier in their approach, where the sampling operator cannot be pushed through a join, i.e., \( \text{sample}(R \bowtie S) \neq \text{sample}(R) \bowtie \text{sample}(S) \).

Zhao et al. [2018] revisied this classic problem, and extended the ideas of Chaudhuri et al. [1999] in several nontrivial directions:

1. They proposed a general framework for random sampling over multi-way joins, which includes the algorithm of Chaudhuri et al. [1999] as a special case.

2. They explore several ways to instantiate their framework, depending on what prior information is available about the underlying data, and offer different trade-offs between sample generation latency and throughput.

They also analyzed the properties of different instantiations and evaluate them against the baseline methods; the results clearly demonstrate the superiority of their new techniques.

In this project, we reproduce the methods in [Zhao et al., 2018] with TPC-H dataset. In Section 2, we summarize the algorithms we implemented by describing the main idea of each algorithm. Our experimental study is in Section 3, and we provide the details of data sets, implementation language, techniques, choices, parameters, experimental setup, measurements, and metrics. We also offer an analysis of our results and compare them with the results in the original paper in Section 3. In Section 4, we verify that the samples returned by each method are indeed uniform by doing Kolmogorov-Smirnov test.

2 Summary of Implemented Algorithms

In this section, we summarize the algorithms we implemented. We have implemented the Reverse Sample for Query 3 (Q3), and Extended Olken and Exact Weight for both Query 3 and Query X (QX). Note that, the Extended Olken we used algorithm we used in this project is actually Extended Olken Basic, but we use Extended Olken to denote Extended Olken Basic in over this paper. The descriptions of these methods are as follows.
2.1 Reverse Sample

The Reverse Sampling (RS) method does sampling from only $R_n$ and traverse backwards to $R_1$, using a sampled record from $R_n$, to produce a sample of the join. It only works for acyclic foreign-key joins with a single source relation, where $R_n$ is called the source relation. In the hyper-graph terminology, such a join must satisfy the following requirements (possibly after relabeling the relations):

1. $\forall i<j$ with $A(R_i) \cap A(R_j) \neq \emptyset$, their common attribute is a primary key in $R_i$ and a foreign key in $R_j$.
2. $\forall i<n−1$, there exists a $j > i$ such that $A(R_i) \cap A(R_j) \neq \emptyset$.

A strong implication of these two requirements is that there is a one-to-one mapping between $R_n$ and the join results, so $R_n$ is called the source relation. Hence, sampling from the join results reduces to just sampling from $R_n$.

In our experimental study, this method that is only tested in Query 3 in this project, since Query X does not satisfy its requirements.

2.2 Extended Olken

We use Python 3.6 and follow the chain sampling algorithm 1 in Zhao’s work [Zhao et al., 2018] to implement our experiment. The Extended Olken (EO) algorithm is implemented as follows.

1. Find $M_{A_i}(R_i) = \max_v d_{A_i}(v, R_i)$ or max frequency of attribute in each input relation $R_i$. Take Q3 for example, $M_{A_2}(R_2) = 41, M_{A_3}(R_3) = 7$

2. Calculate $W(t)$ based on the definition $W(t) = \prod_{j=i+1}^{n} M_{A_j}(R_j)$, $t \in R_i$. Take Q3 for example, $W(t) = 287, t \in R_1; W(t) = 7, t \in R_2; W(t) = 1, t \in R_3$.

3. Build a value to index dictionary. The idea is that if $R_i$ and $R_{i+1}$ join on attribute $A$, for each unique value $v \in R_i(A)$, we find all the indexes $\{id\}$ that at $id$, $R_{i+1}(A) = v$. This dictionary is designed to find right join tuple’s index.

4. Run algorithm. Given a tuple $t$, we first find value of attribute $A_i$, $v$. With $v$, we can find indexes of $t \times R_i$ from the dictionary we build in step 3 $\{id\}$. Since $W(t)$ is constant for $R_i$, $W(t \times R_i) = n \times W(t_{\text{new}})$, $t_{\text{new}} \in R_i$ where $n$ is the number of right join tuples. Then we reject with reject probability. Once accepted, we sample one index from $\{id\}$ then feed the new $t$ back into the algorithm.

Low Efficiency Implementation. At beginning, we didn’t build the dictionary in step 3. We make $t \times R_i$ by searching $R_i$ everytime. We sample a tuple by pandas.sample, which is a really slow process, rather than sampling from indexes $\{id\}$. With this implementation, it would take us 5.5 hours to sample 1,000,000 joined tuples from Q3 compare to 38s with the above algorithm.

2.3 Exact Weight

The exactly weight (EW) sampling algorithm is a more generalized version of Chaudhuri et al.’s work [Chaudhuri et al., 1999]. We followed the description of the method in Section 4.2 from the work of Zhao et al. [2018] and reproduced it. Here is the steps of EW:

\[\text{Low Efficiency Implementation. At beginning, we didn’t build the dictionary in step 3. We make } t \times R_i \text{ by searching } R_i \text{ everytime. We sample a tuple by pandas.sample, which is a really slow process, rather than sampling from indexes } \{id\}. \text{ With this implementation, it would take us 5.5 hours to sample 1,000,000 joined tuples from Q3 compare to 38s with the above algorithm.}\]
1. Assign weight equals to 1 for each tuple in the last relation of the chain join line.

2. Compute the exact weights for each tuple \( t \) in a relation from the last second one of the chain join line all the way to the first one. The weight of tuple \( t \) is the sum of the weights of all the tuples of the previous relation that can be joined with it. (This is a more practical presentation compared with the hyper-graph representation.)

The EW sampling algorithm sets \( W(t) = w(t) \) for each tuple \( t \) from every relation. Based on the mechanism of EW, the rejection rate is always 0. So in our implementation, we first check the exact weights and make sure our exact weights are right. Then in our sampling loop, we omit the rejection rate computation.

Although we build the index for our implementation and try to do optimization on the logical layer, the results are not as same as the Figure 3 in the paper of Zhao et al. [2018]. Fortunately, after tons of optimization, our final results get a stable line for EX.

Figure 1 shows the results of our optimization steps. Here are some details.
1. We first replace the `pandas.dataframe` structure by the `numpy.array`. As shown in the Figure 1(a), the `pandas.dataframe` (blue line) is almost 10× time consuming in this experiment, compared with others in `numpy.array` implementations.

2. After applying `numpy.array` data structure, we found the results for 1 million samples still not good. We detected every step in the sampling loop, and found that the most time-consuming step was from a function called `choice.random()` to choice the id of the next relation based on a possibility distribution. So we optimized it by using the `map(lambda: ...)` to generate all the random list for each point of hyper-graph path in advance. Then we get the green lines in each subfigure in Figure 1, which mostly did a better job especially in the 1 million choice.

3. Then we found that some generation of some random lists cost a lot, which we considered as an “over optimization”. Then we had a step back, and only chose the random list with big length to do the pre-generation first. After this adjustment on the algorithm, we got the red lines in Figure 1.

Our final results are represented by the red lines in Figure 1, which are almost stable and pretty close to the original results.

3 Experimental Study

In this section, we provide the experimental study of algorithms we implemented. We implement the algorithms and test them in both basic experiments and extended experiments. Our results could match the results in the original paper, and demonstrate the correctness of our implementations.

3.1 Experiment Setup

We reproduced our sampling algorithms in Python 3.6 and performed experiments on a Google Cloud machine running Ubuntu 18.04 LTS with four Inter(R) Xeon(R) CPU processors and 40 GiB memory. All the experiments can fit into the memory including the extension.

Our experimental data set is generated using the standard TPC-H benchmark. We use scale factor of 1 to generate 1 GiB data set for our basic experiment and 10 GiB data set for the extension.

3.2 Basic Experiment

We first provide our results on TPC-H with scale factor is 1.
Figure 2 shows the result of our implemented algorithms (extend olken, extend weight, reverse sampling) over the TPC-H data set with the default scale factor of 1 (rough 1GiB), to test the performance of them when collecting $10^3$, $10^4$, $10^5$, and $10^6$ samples. Note that, the reverse sampling algorithm can only applied to Query 3 (results in 2(a)), and cannot be applied to Query X (results in 2(b)).

For EW and RS, the accept rate is always 100% due to the mechanism of the two sampling algorithms. For EO, the accept rate of Q3 is 0.1334 and accept rate of QX is 0.0937. This is different from original paper. One of the reasons is that [Zhao et al., 2018] embedded AGM bound to predict the size of join results. So we guess their $W(t)$ are different from ours.

In the next part, we provide the analysis of both basic experiment and extension, and also compare our results with the results in the original paper.

### 3.3 Extension
We also extend your evaluation for the algorithms above to TPC-H scale factor 10 (rough 10GiB).
Figure 3 shows the result of our implemented algorithms (extend olken, extend weight, reverse sampling) for collecting $10^3$, $10^4$, $10^5$, and $10^6$ samples. Similar with the the case of scale factor is 1, the reverse sampling algorithm can only applied to Query 3 (results in Figure 3(a)), and cannot be applied to Query X (results in Figure 3(b)). As for the accept rate, EW and RS still can achieve 100%. For EO, our accept rate of Q3 is 0.1264 and accept rate of QX is 0.0913.

Compared with the results with original paper, our results can roughly match their results but a little bit slower. We think the reasons are:

1. The programming language we used is Python 3.6, while the authors of the original paper used C++. We suppose the difference of programming languages is the main reason of difference.

2. The CPU of the machine we used to implement our algorithm has lower clock speed than theirs.

We also analyze the differences between the case of the scale factor is 1 and scale factor is 10 as follows:

1. For the algorithm of Reverse Sampling: The running times are almost same for scale factor is 1 and the scale factor is 10, and the case of the scale factor is 10 only used a little more time than the case of the scale factor is 1. The reason is we have built indexes before our algorithm in our implementation. Since that, the time used for join will only be related to the number of samples we collected, not related to the scale of the dataset. That makes the running time almost the same for those two cases. The only difference is the size of indexes and the range of sampling, which causes those tiny difference between these two cases.

2. For the algorithm of Extended Olken: Sampling time of 1 million from Q3 with scale factor 1 and 10 has no significant difference. However, sampling time of 1 million from QX with factor 10 is almost 10 times larger than that of factor 1. This is because reject rate of QX is 40% higher than that of Q3. QX also has 2 more relations than Q3.

3. For the algorithm of Extended Weight: The time consuming keeps stable for generating every size of sample set in our experiments because the time cost mostly concentrates in the weight
pre-processing part. The sampling time in scale factor of 10 is 10 times larger than that of scale factor 1. This is caused by the weight pre-processing part in scale factor of 10 is 10 times bigger than scale factor of 1. Two relations with super big candidates mainly dominate the time consuming, and the candidates of these two relations are almost 10 times bigger in scale factor of 10.

It was also worth mentioning the are several critical point in our implementation:

1. We have build indexes before algorithms. According to the original paper [Zhao et al., 2018], the total running time does not include the building times of all the indexes. That can significantly reduce the total running time.

2. Pandas is a commonly used package in Python for the processing of data frame, and we used it to load with TPC-H dataset. However, we transited the data type from pandas data frame to numpy array (or python list) in the sampling process. We found it empirically that we can reduce the total running time by several orders of magnitude using this way.

4 Kolmogorov-Smirnov Test

Kolmogorov-Smirnov (KS) test is a nonparametric test of the equality of continuous, one-dimensional probability distributions that can be used to compare a sample with a reference probability distribution (one-sample K–S test), or to compare two samples (two-sample K–S test) [Wikipedia contributors, 2018]. We applied the KS test to validate that our samples are uniform. The KS tests were performed on Query X where we use the $\alpha = 0.01$ significance level. We compute the K-S score $d$, the largest distance between the expected cumulative distribution and our experimental cumulative distribution when the sample size equals 1 million.

The results show that our EW sampling algorithm can pass the KS test at the $\alpha = 0.01$ significance level where $d$ should be less than 0.00163, while our EO sampling algorithm failed with a little bit error.

4.1 Order Computation

To compute the K-S scored, we should assign a sorted order on the full join result, and find the position for each sampled tuple in the order. However, materializing the full join for Query X is both time and space consuming (2.4 billion and 239.94 billion tuples generated for scale factor = 1 and 10 respectively). Hence, we take use of the weights and the index information in the EW to (1) construct a specific order for the full join result and (2) compute the position for each sampled tuple in the order.

First of all, our sampled tuples for Query X are recorded as a list of tuples. Each tuple contains five integer numbers. Each number in a tuple represent its position (or id) in its correlated relation.

To compute the order information for the full join result, we first generate five vectors, which we call cumulative weight vectors, for all the relations to store the partial position information for each tuple. Specifically, to compute the cumulative weight for a tuple $t$ in one relation $R$, we first load a list from our pre-computed index (assume the index is on attribute $B$). The list contains all the ids in the $R$ that share the same value with $t$ in $R.B$, which also includes the id of tuple $t$. Then we
use the sum of the exact weights on all the tuples whose position in the list are ahead of tuple $t$.

To compute the ordered position of a sampled tuple, we first map the five ids in the sampled tuple to five cumulative weight values in the corresponded cumulative weight vectors. Then the ordered position for the sampled tuple is the sum of these five cumulative weights values.

### 4.2 KS Test Results

The experimental value of $d$ from Query X with EO and EW sampling algorithms is shown in Table 1. All of the K-S values are small (less than 0.009). The sampled results from EW sampling algorithms can pass the KS test for both the scale factors of 1 and 10. While the results from our EO sampling algorithms are failed with a little bit error.

<table>
<thead>
<tr>
<th>Sampling Algorithm</th>
<th>Experimental d</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF=1</td>
<td></td>
</tr>
<tr>
<td>Extended Olken</td>
<td>0.008591</td>
</tr>
<tr>
<td>Exact Weight</td>
<td>0.001999</td>
</tr>
<tr>
<td>SF=10</td>
<td></td>
</tr>
<tr>
<td>Extended Olken</td>
<td>0.001047</td>
</tr>
<tr>
<td>Exact Weight</td>
<td>0.000444</td>
</tr>
</tbody>
</table>

Table 1: K-S values, where $n = 1$ million

As shown in Figure 4, we have generated four plots showing the experimental results along with the expected uniformed distribution when samples are collected from QX using both EO and EW with scale factors of 1 and 10. From the Figure 4(a) and Figure 4(b), we can see that the cumulative distribution function plots for EO sampling algorithm are still pretty close to uniformed distribution even though they failed to pass the KS test.
Figure 4: KS test of four implementation

5 Conclusion

This project revisits the classic and important problem of join sampling by reproducing the paper [Zhao et al., 2018]. We implemented the algorithms of Reverse Sample, Extended Olken and Exact Weight, and tested them in the TPC-H dataset, and analyzed our results. We also did the Kolmogorov-Smirnov (KS) test to verify the uniformity of our sampling.

References
