CS 645 Project Report
Sample Join

Corey Clemente, Ninad Khargonkar, Sreenivas Venkobarao

December 20, 2018

1 Introduction

Join operations are one of the most commonly used operations in database systems. They can be quite expensive to compute when dealing with multiple relations over a large amount of data. This issue of expensive computation has motivated research around sampling from joins. Most of the literature has focused on the problem of returning a random sample from the full join result. An important fact is that the sampling operation cannot be passed through the join operation which necessitates the requirement of sophisticated algorithms which leverage the information from the pre-computed statistics on the relations. The work in the paper [1] extends previous work on sampling from a two relation join to multi-relation joins and introduces a general framework for sampling which subsumes previous different approaches to sampling as special instances within the framework. In our project we focus on three sampling algorithms described in the paper: Reverse Sample, Exact Weight and Extended Olkens. We perform our experiments on two specific queries used in the paper and also try two extensions given in the project requirements.

1.1 Reverse Sample

This algorithm assumes an acyclic foreign-key join with a single "source" relation. It was shown that there exists a one-to-one mapping between the source relation and the full-join results and hence it is sufficient just to sample from the source relation and reverse trace through the chain of relations to obtain a sample from the full-join. However the requirement of a source relation is a quite strong condition and hence other methods were investigated for sampling.

1.2 General Framework

To deal with the general case, the authors propose a join sampling framework and an associated general sampling algorithm for chain joins (Algorithm-1 in the paper). The algorithm makes use of estimates (weights) for the cardinality of the join size to accept or reject samples with appropriate probability. Since the true information about the join-size is not available, we use these estimates as a proxy for it. The "closesness" of such estimates is controlled by the specific method we are using to initialize these weights. The estimates are defined very broadly

to capture their initializations from different approaches towards sampling. Three constraints are enforced on the estimates during their computation (for any intialization method) to require consistency in the sampling algorithm. The essential fact is that the weight estimates always upper bound the true value and since this estimate is not always 'tight', we introduce accept/reject probability for sampling a tuple from the join result.

1.2.1 Extended Olkens

Extended Olkens as the name suggests is a way of extending the basic algorithm on a two relation join. For the two relation case, it just samples a tuple at random from the first relation and then amongst all tuples joining with it in the second relation, it samples a tuple at random with a weighted probability (the denominator is the maximum frequency on the join attribute). So for extending this, a similar approach is taken where the weighting factor for a tuple in a relation is the product on the max-freqs on the join attributes over all the downstream relations.

1.2.2 Exact Weight

Exact Weight is an extension of Chaudhari's scheme for sampling from a two relation join. This approach samples the first tuple with probability proportional to matching tuples in the second relation (instead of a uniform sample from the first relation). Then amongst all matching tuples in the second relation we accept any one uniformly. To extend it to multiple relations, the authors propose to initialize the estimates with the true values of join-size which is computable in linear time in the size of all relations using a dynamic programming algorithm.

2 Datasets

We use TPCH benchmark dataset to carry out our experiments. It is a decision support benchmark and has multiple tables. It comes with a table generation tool which allows for variable sized tables for a given scale factor. By using a scale factor of 0.1, we obtain a 100MB dataset which we use for initial testing. Eventually we run the same experiments using a scale factor of 1.0 which results in an approximately 1GB sized data set.

3 Implementation

We implemented all of our experiment code in Python2 and use the external Numpy library. Hashing is done with the standard library, hashlib using its SHA-256 implementation. Dictionary and list data structures of Python were used extensively for storage and fast data access. The code structure is as follows:

3.1 Relation

Relations are instantiated as objects of a "Table" class with many associated helper methods and an index on the columns. Any table from the .tbl file on disk is loaded in and an object from the Table and its data and index attributes populated during the loading process. The two

main attributes in the Table class are the data and index both of whom are dictionaries. The keys of the data dictionary are the column names of the table. Then going one level deeper, for each column (key) the value is a list of actual values for that column in the table. Consistent ordering is maintained across the lists for different columns so that a particular tuple is referred to by the same index (iteration number) in across all the lists for columns. The helper methods include a function which gets the maximum frequency of a specific column since this value is used in the weight computation.

3.2 Index

For the index (dictionary) over the columns, its keys are again the columns with the value for each column again being a dictionary where the keys are the unique instances for that column. And for a specific instance for the column, we then have a list of all tuple indices whose column value equals that specific instance. We also have helper methods for the index on the columns which allow traversing through the index without explicitly accessing the dictionary.
After structuring the code for tables in the above manner, it becomes quite easy to logically think about the relations and their statistics along with associated index.

3.3 Weight computation

In the weight $W(t)$ computation part, we divide each specific method (Generalizing-Olken, Exact-Weight, Extended-Olken) into an individual class file. However across all the classes, the methods and their parameters are kept consistent to eliminate duplication of effort while using the sampling algorithm. At the start of the algorithm it just instantiates a weight computation object from the desired class and is oblivious to the actual method going forward i.e the object is treated as black-box which spits out weights for tuples and the semi-joins. In the class file (for a specific method), we run the actual weight computation part during its *init* and cache out the results to a data structure so that retireval is constant-time. This is especially useful for Extended-Olkens method where the weight of all tuples in a relation is the same and hence the space usage is also quite low. The two important methods in the class file are *get_tuple_weight* and *get_relation_weight* which compute the tuple weights and semi-join weights respectively.

3.4 Sampling Algorithm

We implement the sampling algorithm (algorithm-1 in the paper) in the separate file with the method *sampler* being the one returning the sampled tuples from the join after accepting the desired number of tuples and the tables and join columns. It returns a list of lists with each inner list being a single sample from the join and containing the indices of the tuples from the tables for its data. Wherever its required to sample from a discrete (but not uniform) distribution over a support set of size N, we use a simple rejection sampling based algorithm with the proposal distribution being the uniform discrete distribution ($\Pr = \frac{1}{N}$) with the scaling factor being equal to N. This way we avoid instantiating the full vector of probabilities and in the process being space efficient but sacrifice a little bit on the time-cost front. To work around the notion of r_0 tuple introduced in the paper (r_0 is defined such that r_0 joins with everything

in R_1), in essence we just set the rejection probability to zero in the first iteration of the loop and then go ahead with the sampling process from R_1. We also make extensive use of indexes that we built on table columns (while instantiating the relations) for relatively computation in the weights and sampling from the semi-join of matching tuples.

3.5 Parallelization

In order to more efficiently sample, we implemented the PyTorch library to be able to run individual workers to sample independently of each other. Once the data was loaded in our in-memory database, the algorithm to sample a single tuple was distributed amongst several threads. These threads then logged their results to disk, and we read from these log files for our analysis.

3.6 Kolmogorov-Smirnoff Test

The Kolmogorov-Smirnov (KS) test is used to verify whether the samples returned by the sampler using any weight computation method are uniform or not since by theory we expect them to be uniform in nature. We reproduce the tests in the paper for getting out the plots. Specifically we get 1 million samples from Query X and plot out the cumulative distribution function. Since the cumulative distribution function for a continuous uniform probability distribution is a straight line ($y = x$), we expect the cdf from samples also to be the same. The largest absolute difference between the expected and empirical cdf values is defined to be the KS-score which we denote by d. Its like a measure on the "distance" between the two functions.

The cdf for a given sample M is calculated as follows. First we need to decide on an ordering to sort two tuples for their relative position in the full join J . We can use any ordering as long its consistent and deterministic. To achieve this we hash the tuples using the SHA-256 algorithm and then to map the hexadecimal value to an integer (referred to as $h(t)$ for a given tuple t) in the interval $[0, |J|]$, we just take the modulus of that integer with $|J|$. This way we get the relative position in the full join result without explicitly computing it since the full join takes up too much space. Now once the ordering is fixed to reproduce the plot, we sort all the tuples in the sample M and then the empirical distribution (y value) for a tuple at the ith position will be $\frac{i}{|M|}$. Also its relative position in the x-axis for a tuple t is given by $h(t)$.

4 Extensions

4.1 Extended Olkens - AGM Bound

Apart from generalizing Olken's algorithm to any amount of chain joins, you can also bound the upper limit of a join size using another result known as the AGM bound.

For chain joins, for a $t \in R_i$, just like in generalizing-olkens we set its value to a bound on the following quantity: $|R_i \bowtie ... \bowtie R_n|$ where n is the number of tables in the join. The key observation is that the weight is same for all tuples in a relation. The bound comes from the

special case of chain joins in the AGM bound:

$$|R_i \bowtie ... \bowtie R_n| \leq \min_{I:\{i,n\}\subseteq I, I\cap\{j,j+1\}\neq\emptyset, j=i+1,...,n-2} \prod_{k\in I} |R_k|$$

Here, consider all sets of tables after R_i in the join that include R_{i+1} and R_n, as well as at least one table from every intermediary join. For each of these sets, compute the multiplication of table sizes. The minimum of these sizes becomes the upper bound on the entire join size.

Following the interpretation, we built a two-step algorithm to generate I efficiently for each table in the join.

First, we organized the join as a binary tree, where for each intermediary join pair, you can choose either R_j or R_{j+1} (left or right child) to satisfy the condition. Upon finding all possible path from the root node to a leaf node, you have actually found all possible subsets of tables that satisfy the condition in the minimum operation. Filter out any duplicate paths found. Call this set S.

Second, with this filtered set, find all entries such that there exist another entry which is a subset:

$$L = \{s_2 | s_1, s_2 \in S, s_2 \neq s_2, s_1 \subset s_2\}$$

Everything in L is guaranteed to not be the minimum of the AGM equation since there is a subset that contains some (but not all) of the same tables. We are multiplying lengths of the tables, so everything in L is guaranteed to be larger than everything in S/L. These can be removed from the final set that must be considered.

Repeating this process for each table in the join will produce a list of table subsets that need to be checked. We can pre-compute the minimum across each subset during initialization, so the call to the weight function becomes constant time.

5 Results

Some of our results differ significantly from the results demonstrated in the original paper. We believe this to be due to the difference in scale factor, and possibly due to the use of Python instead of C++ as was done by the original authors.

Exact Weight: We observe a linear increase in time taken per sample for exact weight algorithm in both the queries. The authors of [1] report a constant time per sample. Since our implementation was un-optimized, and reports the time per sample by polling the system clock in a multi-threaded setting, we think the time recorded might not be accurate.

As sanity checks, we evaluated the size of the full join for Query 3. This was compared with

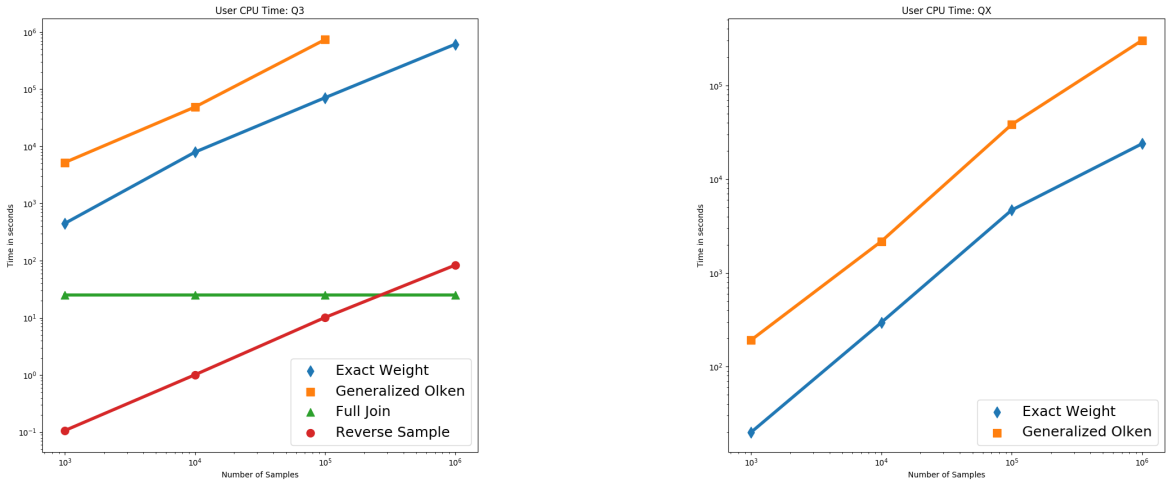

Figure 1: TPC-H sampling collection time for scale factor 1. Left plot refers to Query-3 while the right plot refers to Query-X

the $w(r_0)$ (tuple weights of a canonical tuple in exact weight algorithm) of the first relation in the Query 3 join. These numbers were exactly the same. So we're confident that the weight computation was implemented correctly, and that the numbers look different because our code is unoptimized.

Specifically, to compute $w(t \rtimes R_i)$, we sum over arbitrary indexes in a python list, which is known to be slow, and could have been replaced by numpy slicing.

For Query 3, Full Join computation took 25.75 seconds on an Ubuntu machine with 8GB RAM, at scale factor $= 1$.

The exact weight computation completed in 1 second. For query 3, we verified that $w(r_0) = 6001215$ was the same as the number of rows from the full-join. Additionally, we evaluated that $w(r_0) = 2400301184$ i.e., a little more than 2 billion rows for query X.

Generalized Olken: Here, Generalized Olken appears to be following a linear trend in both queries. The authors of the paper used the AGM (for both chain and non-chain joins) version of Extended Olkens in their graphs, and state that results obtained for upper bounds from Generalized Olkens and Extended Olkens are not generally comparable. The timings we report cannnot be compared to the times of their findings.

Extended Olken (AGM bound) For our AGM weighting analysis, we were unable to get enough samples out in order to run experiments. While we got several samples using the

weighting scheme, the acceptance probability was too low for the algorithm to run quickly. This could indicate a misunderstanding of the algorithm as stated in the paper, although we believe we followed the mathematical definition correctly for generating the upper bound. This could arise from an incorrect computation of the bound when dealing with corner cases of the subset of relations to consider.

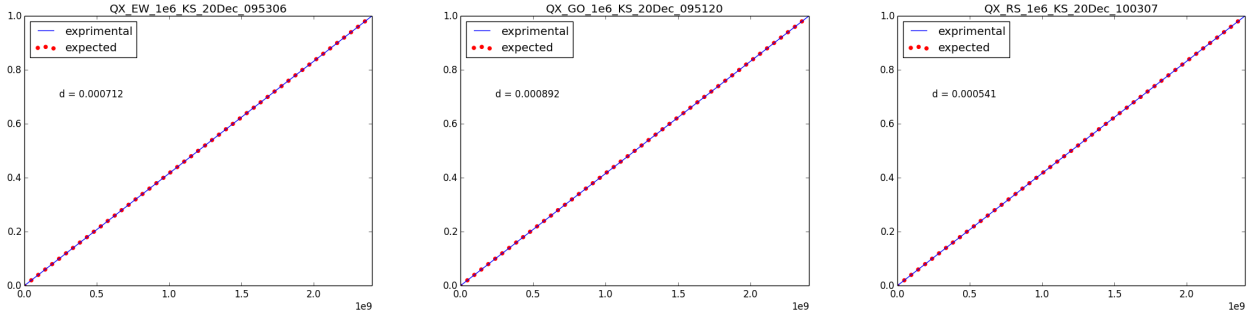

Figure 2: KS test cdf plot for Query-X using Exact Weight, Generalized Olkens and Reverse Sampling.

The KS-score plot in figure 2 show the expected linear trend ($y = x$) for all three methods tested. Also the values for the KS scores on Query-X are shown in the table below. As we can observe, the values are very small which signifies that the samples returned by the algorithm are indeed uniform in nature.

Method	KS-score (d value)
Exact-Weight	0.000712
Generalizing-Olken	0.000892
Reverse-Sample	0.000541

6 Conclusions

Our key take-aways from this project was learning to design and implement a database such that we were able to use indexes optimally. We also implemented several algorithms for sampling from joins, and investigated their performance. Due to resource constraints and implementation differences, we are unable to match the results reported by [1].

References

[1] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi, "Random Sampling over Joins Revisited," (Houston, TX, USA), pp. 1525–1539, ACM Press, 2018.