Scalable Reservoir Sampling on Many-Core CPUs

Altan Birler
Technical University of Munich
altan.birler@tum.de

ACM Reference Format:
Altan Birler. 2019. Scalable Reservoir Sampling on Many-Core CPUs. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3299869.3300096

1 PROBLEM AND MOTIVATION

Database systems need to be able to convert queries to efficient execution plans. As recent research has shown, correctly estimating cardinalities of subqueries is an important factor in the efficiency of the resulting plans [7, 8]. Many algorithms have been proposed in literature that utilize a random sample to estimate cardinalities [6, 9, 13]. Thus, some modern database systems choose to store a materialized uniformly random sample for their relations [3, 6]. Such samples are built and refreshed when statistics are gathered, by loading uniformly random tuples from the relation in disk using random IO. Drawing a random sample from disk storage can take multiple seconds which is highly prohibitive. To reduce these costs, the frequency at which the samples are refreshed is lowered, leading to stale samples and worse cardinality estimations, until the samples are refreshed again. Maintaining the samples online, during the insert/update/delete operations, simply avoids these costs while keeping the sample fresh. Furthermore, the improvements in multi-core performance far surpass the improvements in single-core performance and utilizing this trend is essential to achieving new leaps in performance and efficiency [2, 11]. Therefore, we propose a highly-scalable shared-memory approach to online sampling that utilizes a single global sample and a global work-sharing structure to distribute the sampling work among the threads (Figure 1).

2 BACKGROUND AND RELATED WORK

Reservoir sampling is used for online uniform sampling from an infinite source [12]. The algorithm first fills the sample with the initial tuples. After the sample has been filled, for every new tuple, a probabilistic decision is made whether or not to insert the tuple into the sample. This requires a new random number to be generated for every tuple. Vitter [12] describes an important optimization: Instead of making a decision for every new tuple, one can probabilistically generate the number of tuples that will be consecutively skipped until a tuple has to be inserted into the sample. Instead of producing the decisions *skip, skip, skip, insert, skip, skip, insert, insert*, the algorithm produces the values 3, 2 and 0, corresponding to the distances between *insert*s. The expected number of tuples that will be inserted into the reservoir is given by the formula $O\left(n \cdot \left(1 + log\frac{N}{n}\right)\right)$ where n is the size of the sample and N is the total number of tuples [12]. This means that the number of tuples inserted into the sample and thus the number of *skip lengths* required is much less than the total number of tuples.

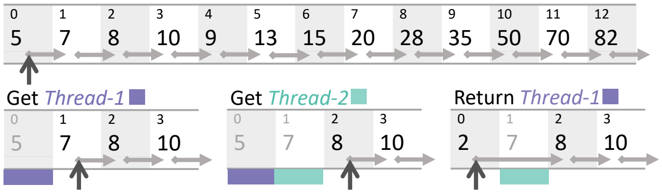

Figure 1: Skip Source

Parallelizing Vitter's reservoir sampling algorithm is essential for utilizing modern hardware developments and supporting modern database architectures. Methods have been proposed for parallelizing reservoir sampling [1, 10]. However, these proposals mainly focus on maintaining independent samples (different reservoirs) for each processing unit and then merging these samples later. Sanders et. al. [10] propose a way to merge these samples using the hypergeometric distribution. If every unit receives the same amount of work, the resulting sample can be as large as the sum of the sizes of the local samples. If units receive greatly different amounts of work, which does happen in OLTP settings, the resulting sample becomes much smaller, even if one does not require the sample to be perfectly uniform. For example, if there are two local samples of size 500, and one receives 0 elements while the other receives all the rest, it is impossible to construct a final sample of size 1000, the sum of the sizes of the local samples. Al-Kateb et. al. [1] describe a way to increase the sample size while keeping the probability of non-uniformity below a certain threshold. Even without the issue of decreasing the size of the sample, requiring local samples to be generated and then merged complicates things

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

for an OLTP system, where new threads with arbitrary workloads may start and end at arbitrary times and the degree of parallelism may elastically change [5].

3 APPROACH

In our approach we sought to develop a multi-threaded system where a single global sample is used per relation. By maintaining a *single* global sample we avoid the problems with using multiple reservoirs, most importantly the requirement to merge reservoirs.

We take Algorithm Z, the most efficient variant of reservoir sampling, as described by Vitter [12], and interpret it as the generation of an infinite list of *skip lengths*, and the sequential execution of these *skip lengths*. If a *skip length* contains the value m, its execution corresponds to skipping the upcoming m tuples of the input stream and inserting tuple $m + 1$ into the sample at a random position.

In multi-threaded systems, every thread comes with its own input stream, which has an unknown length. The idea is to assign the *skip lengths* among the threads in an efficient manner. This corresponds to weaving the substreams of data we have into a single global stream, which is then sampled from using Algorithm Z. Looking at Figure 1, assigning the 0th *skip length* to Thread 1 corresponds to ordering the tuples 0-4 of substream 1 at the places 0-4 of the global stream. Similarly, if we then assign the 1st *skip length* to Thread 2, we order the tuples 0-6 of substream 2 at the places 5-11 of the global stream. After receiving a *skip length*, the threads perform the skip and insertion operations in parallel.

When inserting a tuple into the sample, a thread picks a random location and tries to overwrite the existing tuple at that location with the new one. Note that the existing tuple may only be replaced if its conceptual order in the global stream is before the order of the new tuple. If the global order of the existing tuple is higher, the new tuple must be discarded. To ensure this, threads have to store the global order when inserting a tuple into the sample and check this value while overwriting existing tuples. This allows the execution of our algorithm to be interpreted as a single-threaded execution of Algorithm Z.

There is a global structure called the Skip Source (Figure 1) that is used to distribute skips. When a thread requires a *skip length* it asks for it from the Skip Source, and when a thread is done, it returns the remaining number of skips back to the Skip Source. The main source of contention is the access to the Skip Source data structure, which stays minimal due to the fact that the number of skips needed increases logarithmically as the dataset grows larger [12]. Details of the algorithm are in explained in Appendix A.

Our implementation currently only supports inserts. However, updates and deletes can also be implemented. Updating a tuple requires searching for it in the sample and updating the entry if it exists. For deleting tuples, tuples inserted in the future can be *reordered* in their place. If the deleted tuple was in the sample, a new insert replaces the old tuple, if not, we make an additional skip. After all deleted tuples are *replaced*, the sample stays uniform under the assumption that the probability of a tuple being deleted is independent from that tuple being in the sample.

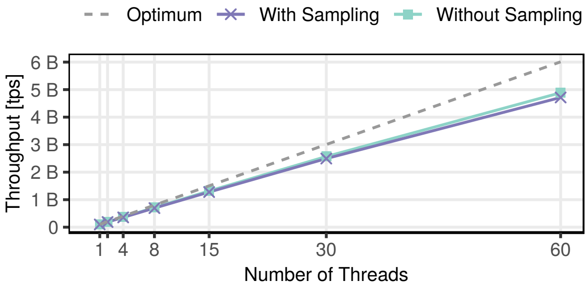

Figure 2: Throughput of the algorithm

4 EVALUATION

We evaluated our algorithm on a machine with 4 NUMA nodes of Intel Xeon E7-4870 v2 CPUs with 15 cores (30 hyperthreads) each. Our dataset is a text file containing 2^{32} strings of length between 3 and 13 bytes. The dataset is first loaded into memory. The threads then receive evenly sized chunks of the data and are tasked with iterating through the given chunk loaded on their local NUMA node and running the sampling logic. These chunks contain the same amount of bytes but differ in the amount of tuples. The iteration is done by finding tuple separator characters one after the other using SIMD instructions. We measure the throughput (tuples per second) of iterating through (and sampling) the entire dataset with varying number of threads.

The results are shown in Figure 2. We compare *without sampling*, which only iterates through the data and extracts tuples without any processing or sampling, with *with sampling*, which runs the sampling logic on all the extracted tuples. The *optimum* is a theoretical line, where the slope is equal to the throughput of an efficient single-threaded implementation of Algorithm Z.

The overhead of sampling over simply iterating through the data without sampling is less than 3.6% across all threads which supports our claim that the synchronization overhead is minimal. Note that in an actual database, the tuples would undergo further processing such as being appended to the relation in storage and inserted into index structures. All additional processing would decrease the relative overhead of sampling. Thus, our algorithm is an efficient and effective replacement to re-scanning relations to reconstruct samples.

ACKNOWLEDGEMENTS

We are indebted to Bernhard Radke and Thomas Neumann for their encouragement and invaluable guidance throughout this project.

A ALGORITHM DETAILS

Skip lengths are distributed among the threads using the Skip Source structure, which is a linked-list of *skip lengths*. Every *skip length* in the Skip Source contains a pointer to the next *skip length* that will be executed after this *skip length*. There is also a single current_index pointer that points to the *skip length* that will be assigned to the upcoming thread that runs the GetSkip operation (Figure 3). A thread acquires a new skip by popping a *skip length* from the front of the Skip Source using the GetSkip operation. When a thread is done it pushes its remaining number of skips as a single *skip length* to the front of the Skip Source by using the ReturnSkip operation (Figure 3). Both operations must be atomic and thread-safe as they can be used concurrently.

The GetSkip operation (Figure 3) acquires a *skip length* from the Skip Source for the current thread. This operation moves the current_index pointer to the position of the upcoming *skip length* (Figure 1). To ensure thread-safety, current_index or next_skip (*pointers[current_index]*) must not be modified by other threads while updating current_index. Here, one can either use a DCAS operation [4], or a load-linked/store-conditional (LL/SC) loop on current_index due to the fact that current_index has to have been modified for next_skip to have been modified. We prefer the LL/SC construction, implemented on x86 hardware by using CAS and the higher-order bits of current_index as a version counter. When a thread starts, it acquires a *skip length* by running a GetSkip operation which it then stores locally. At every insert, this value is decreased. When the local_value reaches 0, the next inserted tuple is also put into the sample at a random position if its version is higher than the version of the tuple already there. When a thread is done, it needs to store the remaining local_value back in the Skip Source so that the remaining skips can be done by another thread. For this, the ReturnSkip operation (Figure 3) is used. We push the local_index into the front of the linked-list of skips by setting *pointers[local_index]* to the current head of the list (*ci*) and setting the head (current_index) to local_index.

GetSkip and ReturnSkip are the main sources of contention in our algorithm. However, the number of times these operations are performed is $O\left(n\left(1+log\frac{N}{n}\right)+2\cdot\#\text{threads}\right)$. For 2^{64} tuples, a sample size of 1024, and 128 threads, less than 40,256 operations are expected which is much less than the total number of tuples processed. The synchronization overhead is thus minimal.

```
GetSkip Operation
1: $ci \leftarrow LL(current\_index)$
2: $SC(current\_index, pointers[ci])$ {goto line 1 if fails}
3: $local\_value \leftarrow skips[ci]$
4: $local\_index \leftarrow ci$

ReturnSkip Operation
1: $skips[local\_index] \leftarrow local\_value$
2: $ci \leftarrow LL(current\_index)$
3: $pointers[local\_index] \leftarrow ci$
4: $SC(current\_index, local\_index)$ {goto line 2 if fails}
```

Figure 3: GetSkip & ReturnSkip Operations

REFERENCES

[1] Mohammed Al-Kateb, Byung Suk Lee, and Xiaoyang Sean Wang. 2007. Adaptive-Size Reservoir Sampling over Data Streams. In *SSDBM*. IEEE Computer Society, 22.

[2] Gustavo Alonso. 2013. Hardware killed the software star. In *ICDE*. IEEE Computer Society, 1–4.

[3] Surajit Chaudhuri, Eric Christensen, Goetz Graefe, Vivek R. Narasayya, and Michael J. Zwilling. 1999. Self-Tuning Technology in Microsoft SQL Server. *IEEE Data Eng. Bull.* 22, 2 (1999), 20–26.

[4] Michael Greenwald. 1999. *Non-blocking Synchronization and System Design*. Technical Report. Stanford, CA, USA.

[5] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[6] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

[7] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[8] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB* 2, 1 (2009), 982–993.

[9] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *PVLDB* 11, 9 (2018), 1016–1028.

[10] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. 2018. Efficient Parallel Random Sampling - Vectorized, Cache-Efficient, and Online. *ACM Trans. Math. Softw.* 44, 3 (2018), 29:1–29:14.

[11] Herb Sutter. 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal* 30, 3 (2005), 202–210. http://www.gotw.ca/publications/concurrency-ddj.htm

[12] Jeffrey Scott Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (1985), 37–57.

[13] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899