Concurrent Online Sampling for All, for Free

Altan Birler
TUM
altan.birler@tum.de

Bernhard Radke
TUM
radke@in.tum.de

Thomas Neumann
TUM
neumann@in.tum.de

ABSTRACT

Database systems rely upon statistical synopses for cardinality estimation. A very versatile and powerful method for estimation purposes is to maintain a random sample of the data. However, drawing a random sample of an existing data set is quite expensive due to the resulting random access pattern, and the sample will get stale over time. It is much more attractive to use online sampling, such that a fresh sample is available at all times, without additional data accesses. While clearly superior from a theoretical perspective, it was not clear how to efficiently integrate online sampling into a database system with high concurrent update and query load.

We introduce a novel highly scalable online sampling strategy that allows for sample maintenance with minimal overhead. We can trade off strict freshness guarantees for a significant boost in performance in many-core shared memory scenarios, which is ideal for estimation purposes. We show that by replacing the traditional periodical sample reconstruction in a database system with our online sampling strategy, we get virtually zero overhead in insert performance and completely eliminate the slow random I/O needed for sample construction.

CCS CONCEPTS

• **Information systems → Query optimization**; **Database query processing**; • **Theory of computation → Sketching and sampling**.

KEYWORDS

online sampling, database statistics, query optimization

ACM Reference Format:
Altan Birler, Bernhard Radke, and Thomas Neumann. 2020. Concurrent Online Sampling for All, for Free. In *International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3399666.3399924

1 INTRODUCTION

A random sample of a large dataset is an effective tool to estimate the number of data items satisfying an arbitrary predicate [14, 18]. Many database management systems, thus, maintain samples of base relations as part of their statistics collection. These samples can then be used during query optimization to estimate the selectivity of filter predicates and cardinalities of intermediate results [11, 16]. Accurately estimating these selectivities and cardinalities is vital for the system to generate efficient query execution plans [12, 15, 23].

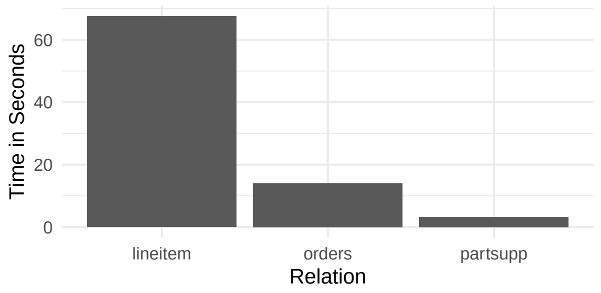

Figure 1: Time to read 1000 random tuples from TPC-H SF5

While querying a small, materialized sample is fast, maintaining the sample usually incurs high costs due to the required random reads. These random reads are of course especially painful in a disk-based system, where e.g. reading 1,000 random tuples can take multiple seconds as can be seen in Figure 1. But even on modern systems, where most of the data resides in main memory, building such a random sample on demand can cause noticeable latency in query execution. To mitigate this latency impact, some systems, such as HyPer [10], accept the sample to become stale and only recalculate it once a certain amount of the underlying data changed.

The cost of periodical sampling can be alleviated by utilizing an online approach where the samples are kept up-to-date while incoming items are processed, avoiding the need to go back and re-read data when the sample is needed. However, an online approach must be able to work within existing software that process data, which raises the requirements for adaptability and performance.

Furthermore, in today's hardware landscape, performance gains are no longer achieved through improved computational power of a single core, but instead through an increased number of cores running in parallel. To fully reap the benefits of modern hardware, software and algorithms must be designed to allow for maximum parallelism, potentially adaptively allocating and deallocating threads depending on the workload. Thus, a sampling approach with a goal of integrating into such systems, must also be capable of adapting to the needs of any parallel workload, scaling as needed.

These requirements lead us to develop a generic online sampling algorithm which

(1) keeps samples up-to-date while processing incoming items,
(2) has negligible overhead in terms of both, per thread work and contention,
(3) uses a constant amount of memory per thread,
(4) eases integration into existing solutions due to a simple interface,
(5) provides a single, continuously available sample of high quality for each relation,
(6) performs a minimum amount of shared memory writes in bulk loading scenarios.

The concurrent online sampling approach we describe is based on Vitter's reservoir sampling [22], where incoming tuples are placed into a reservoir with a probability reciprocally proportional to the total number of items processed so far. A multi-threaded implementation of this algorithm is described in [21]. This implementation guarantees the sample to represent the entirety of the population seen by all threads at any point in time. Due to this guarantee, the synchronization overhead of this implementation increases with the number of collaborating threads. Thus, the key idea of the algorithm proposed in this paper is to relax this requirement on the freshness of the sample to allow for maximum scalability. Our algorithm can construct two types of samples: (1) a sample that is up-to-date, but potentially misses tuples that have not yet been inserted by threads lagging behind or (2) a perfectly correct representation of a subset of the dataset that has been seen up to the point the sample is requested.

The rest of this paper is organized as follows: In Section 2 we discuss the foundations and describe prior work on online sampling. We give an overview over the proposed algorithm in Section 3. After performing a theoretical analysis of the algorithm in Section 4 we show the results of an experimental evaluation in Section 5. Finally we conclude the paper in Section 6.

2 FOUNDATIONS & RELATED WORK

In his seminal work on random sampling from a source of unknown size, Vitter describes reservoir sampling as *Algorithm R* [22]. This algorithm draws a uniform sample S of size m from a data set D of an unknown size n in a single pass. Building a sample is of course only relevant if the data set is larger than the sample. For the rest of this paper we thus assume $n > m$.

To obtain a random sample, any m item subset of D must have the same probability of being S. Vitter achieves this by probabilistically deciding for each processed item of D, whether it has to be placed in the reservoir or not. We call an item to be placed in the reservoir a "reservoir tuple". The set of all reservoir tuples constitutes the "reservoir". We say that an item that has not been placed into the reservoir has been "skipped". The downside of this approach is that it requires a probabilistic decision for each item in D.

In the same work, Vitter also describes an optimal single-threaded sampling algorithm which makes use of the fact that the number of items skipped between two consecutive reservoir tuples follows a geometric distribution. Instead of deciding for each incoming item, whether it has to be placed in the reservoir, the "skip length"s between consecutive reservoir tuples is generated randomly. A "skip length" is the number of consecutively skipped tuples between two tuples that have been consecutively placed in the reservoir. As the expected total number of reservoir tuples for a data set of size n is $O(m \cdot (1 + \log(n/m)))$ [22], the number of discarded items grows exponentially with increasing n. Thus, only $O(m \cdot (1 + \log(n/m)))$ random numbers need to be generated to obtain a sample of size m from a data set of size n.

Li's *Algorithm L* [13] uses a different interpretation of sampling: Each item in D is assigned a uniform random value between 0 and 1. The m tuples with the smallest such random value then constitute the sample. Algorithm L additionally makes use of a compact and efficient computation of the skip lengths.

Vitter's and Li's algorithms allow for efficient online sampling on a single core. To achieve maximum performance on the nowadays omnipresent many-core systems, these algorithms must be empowered to run on multiple cores in parallel. Two principles can be applied to parallelize reservoir sampling. Either multiple independent samples are built per thread and merged into a global sample on demand, or a single, shared sample is maintained and accesses to this sample are synchronized between threads. We call the former the "distributed" setting, whereas the latter is called the "shared" setting. Both approaches are described in literature [1, 20, 21].

The first approach to concurrent online sampling, the distributed setting, keeps local samples for every thread. Its main advantage is that t threads can build their samples locally without communication. The final sample is obtained by merging the local samples. If we require that the final sample has a certain size, all local samples must be of the same size, if there are no guarantees given as to how much data each thread will process. This implies $O(t \cdot m)$ memory use for a sample of size m. Sanders et al. [20] propose merges that result in larger final samples, if one were to accept a certain error bound. Al-Kateb et al. [1] describe a mechanism to adaptively adjust the size of these thread-local samples, shrinking and growing them as needed. Hübschle-Schneider and Sanders [8] communicate information among threads that allows to trim the local set of tuples which are sampled. However, these systems do not an ensure optimal memory bound of $O(m + t)$ and are not designed for situations where threads are established and terminated arbitrarily.

The second approach to concurrent online sampling, the shared setting, keeps a single shared sample, usually maintained by a central coordinator. Note that, even though we call such approaches "shared", they are not restricted to shared memory architectures, but can also be used in shared-nothing environments. While in shared-nothing environments, communication is achieved through message-passing, in shared memory, threads communicate through synchronized reads from and writes to memory shared by all threads.

There has been much research on this kind of sampling [4–6, 21], however, the communication costs of all these algorithms depend on the number of employed threads. Tirthapura and Woodruff [21] prove that the lower bound of expected messages to build a sample of size m with t threads is $\Omega\left(\frac{t}{log(1+t/m)} \cdot \log\left(\frac{n}{m}\right)\right)$. Thus, increased parallelism results in higher communication cost, which ultimately impedes the algorithms scalability to higher thread counts. The dependency on the number of threads is rooted in a strict definition of the correctness of a sample: All proposed algorithms aim to, at any time, provide a correct uniform sample of all tuples processed by all threads so far. By slightly relaxing this requirement, we propose an algorithm optimized for a shared-memory architecture that provides high quality samples and scales perfectly to higher thread counts. Our approach is based on prior work [3], which we improve upon by imposing tighter memory bounds, presenting a better analysis of the quality of the resulting sample, as well as providing a more thorough experimental evaluation.

3 OUR APPROACH

We will now describe how to extend optimal single-threaded reservoir sampling approaches to support scalable concurrent execution efficiently. We assume that the sample has initially been filled with

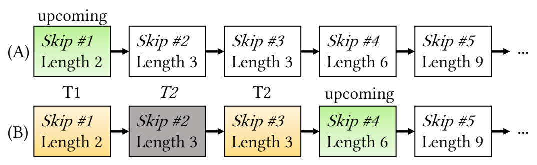

Figure 2: Infinite list of skips

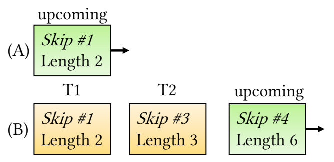

Figure 3: Memory optimized list of skips

the first tuples as described in Section 3.7. Every thread receives an independent sequence of tuples that need to be sampled. The number of tuples each thread will receive is initially unknown, and threads might stop receiving further tuples at any time. The single-threaded approach determines a set of tuples that will be included in the reservoir, the reservoir tuples. Between these reservoir tuples are many tuples that are skipped. With "skip" we denote a contiguous set of tuples that are skipped between two reservoir tuples (or before the first reservoir tuple). A "skip" has a "skip length" which denotes the number of skipped tuples it contains. The data set is split into these skips each followed by a single reservoir tuple. Our approach is to distribute these skips among different threads based on our previous work [3]. After a thread received a skip and its corresponding skip length S_i, it skips the upcoming S_i items without placing them into the sample. This can be done locally without any synchronization. Afterwards, the thread will then place the next incoming tuple into the sample, as the reservoir tuple corresponding to the skip. Such a parallel execution and the generated sample correspond to the execution of the single-threaded algorithm and its sample, where the global ordering of the tuples is determined by how the skips were distributed among the threads.

3.1 List of Skips

Skips to be distributed across multiple threads are stored in a List of Skips (LoS). For the sake of argument, assume that we have an infinite list of skips as shown in Figure 2 row (A). The skip that will be processed next by an upcoming thread, the first one that has not yet been acquired, is marked as *upcoming*. Threads use this list to acquire skips that have not been used yet. In row (B), thread T1 has acquired Skip #1 which it is currently processing. Thread T2 has acquired and finished Skip #2 and already acquired Skip #3 which it is currently processing. The upcoming skip is Skip #4. Looking at the list of skips, it is clear that not every skip is actively utilized. The lists filtered down to the skips that are actually needed can be seen in Figure 3. Here, we only keep the skips currently in use by a thread and one upcoming skip. In row (A), we see that the list only contains a single node, the upcoming node. If a thread were to acquire the upcoming skip, it must first generate the next upcoming skip so that another incoming thread will not find the list empty. In row (B), each of the two threads holds one node and there is still an upcoming node available. Since the total number of nodes excluding the upcoming node is limited by the maximum number of threads, we additionally opt to keep a preallocated free list of nodes which threads can acquire. This allows us to not use any external memory allocation during the execution resulting in more predictable performance characteristics.

3.2 Data Structure

In our implementation both the LoS and the free list are represented by a head pointer, pointing to the node at the head of the respective lists. The nodes within a list are singly linked, where each node contains a successor pointer. Since we reuse nodes, repeatedly removing them from the list and putting them back in, we can encounter the ABA problem [9], where we do not notice that the head pointer has been modified and then changed back. This would mean that the head node could have been modified by another thread, invalidating the information that has been previously read from the head node, such as its successor. To mitigate this, the head pointers also contain a version counter, which is incremented whenever the pointer is modified. So if the head pointer is changed to a different node but afterwards changed back, this can be recognized by other threads noticing that the version has changed. To modify these head pointers, the compare and swap operation is used, where the pointer's value is atomically changed only if it has not been modified since its value has been originally read. We preallocate the nodes, one for each thread and one additional node as the upcoming node, in a contiguous array which can be accessed by a 32-bit offset. Head pointers are stored as 64-bit values where the first 32-bits are the version and the rest is the pointer stored as an offset into preallocated contiguous array of nodes. The structure of LoS is based on the LockFreeStack as described in [7], extended with the close integration of a free-list of preallocated nodes and adapted for the operations required to distribute skips while keeping tight memory bounds, as described in Section 3.4.

A skip node contains information about the skip such as the skip index i, the length of the skip S_i, and the W_i value used by Algorithm L [13] in generating the skip where:

$$\begin{aligned}\text{random}() &:= \text{“A uniformly random number in } [0, 1]\text{”}\\ W_1 &:= e^{\log(\text{random}())/m}\\ S_i &:= \text{floor}(\log(\text{random}())/\log(1 - W_i))\\ W_{i+1} &:= W_i \cdot e^{\log(\text{random}())/m}\end{aligned}$$

Every skip corresponds to a range of tuple indices in the single threaded reservoir sampling. The first skip corresponds to the indices $[0, S_1]$, where the first S_1 tuples are skipped and the last tuple is the reservoir tuple. The second range similarly corresponds to the indices $[S_1 + 1, S_1 + 1 + S_2]$, and the i-th range corresponds to $\left[\sum_{k=1}^{i-1}(S_k + 1), \left(\sum_{k=1}^{i}(S_k + 1)\right) - 1\right]$. The last index within the range of a skip is the index of a reservoir tuple, and the previous indices are indices of tuples to be skipped. Skip nodes only need 17 bytes of space in our implementation. The skip length is an 8 byte unsigned integer, the W value is a 4 byte float, the skip index is a 4 byte unsigned integer, and the pointer to the successor node is stored as a 1 byte index into the preallocated array of nodes. Samples of sizes $< 2^{13}$ require less than 20 bits for skip indices with data sets of size 2^{64} with high probability, so 4 bytes is more than enough for the skip index. In our implementation, we also do not need more than 120 threads, so 1 byte is enough to store the successor node's index. However, more bytes could be used for successor pointers if needed. Nodes easily fit into cache lines which typically contain at least 32 bytes. We align nodes to the cache line size to reduce unnecessary contention between CPU cores.

3.3 Sample Construction

When a thread needs to processes a tuple for sampling, it needs to determine this tuple's position in the global order within the data set. The global order will determine whether or not this tuple is a reservoir tuple. To determine the global order of tuples, a thread acquires a skip i. The skip i will determine the global order of a number of upcoming tuples. The thread's upcoming S_i tuples will be skipped. The next tuple after the skipped tuples is a reservoir tuple and must be placed into the reservoir. When the thread receives no more than k tuples where $k < S_i + 1$, the range $[k + 1, S_i + 1]$ will remain unused and the remaining range must be returned so that another thread can use it to determine global orders.

Before we discuss how skips are acquired and returned, we will address how a sample can be constructed from reservoir tuples. We can handle reservoir tuples in two different ways resulting in two different samples with different properties. The first is to put all reservoir tuples into a shared reservoir, where the i-th reservoir tuple is placed at position i within the reservoir. To generate a sample, we iterate through the reservoir tuples, placing them one by one at uniformly distributed random positions in the sample. Once we encounter the first reservoir index j where no tuple been assigned yet, we will have constructed a correct sample of the tuples globally ordered before the reservoir tuple at $j - 1$. Such a sample is a correct sample of that subset of the data, since the probabilistic properties of the single-threaded reservoir sampling do not depend on the initial order of tuples. This assumes that the probability of a tuple being assigned a particular index is independent of that index being a reservoir index. This is a safe assumption, since unless a thread scheduler adversely tries to schedule threads knowing what skip lengths have been generated, simple differences in the threads processing speeds will have no impact. It is possible for there to be reservoir positions with indices larger than j that have been filled while position j remains empty, since threads could be processing tuples at different rates. However, as more data is processed, such holes will get filled, and it will be possible to find larger and larger indices j where all previous reservoir positions are assigned. So while we do not have a sample that represents the entire data set that has been seen up until the point the sample is queried, the sample will cover more and more of the data set as time goes on.

Another option is to simply ignore unassigned reservoir indices and continue to insert all the reservoir tuples that have been determined. Now the sample potentially represents more of the data, but it is not perfect in a theoretical sense, since it potentially would have different entries if there were no unassigned reservoir positions. There are at most as many such entries as there are threads, since at any time, there is at most one incomplete skip per thread. Since the missing entries are also quite random, the sample is still useful in determining the approximate properties of the data set it represents. If we are fine with generating a sample of this form, we do not need to store the entire reservoir, we can simply store the sample. When a reservoir tuple is encountered, the tuple shall be directly placed at a random position within the sample. To ensure that older tuples do not override newer tuples within the sample, the skip index corresponding to the reservoir tuple must be stored with the tuples themselves. A thread then only replaces a tuple within the sample if the thread's tuple has a higher skip index, which prohibits really slow threads from overwriting the work of threads that have progressed further. To synchronize writing tuples within the sample, a fine grained lock is used per tuple.

Table 1: Actions on the free list when acquiring a skip

Thread / List of Skips	owns node	doesn't own node
contains one skip node	-	allocate
contains more skip nodes	deallocate	-

3.4 Skip Acquisition

We will now discuss how threads acquire and return skips from the LoS. After a thread acquires a skip node, it can use the information on the node to determine reservoir tuples locally. The LoS initially contains only one node, the initial, first skip. Our algorithm has four properties (**P.1**-**P.4**) that ensure that no data races take place and that tight memory bounds are held:

P.1 During execution, the LoS always contains at least one skip node, so that the upcoming thread can always receive a skip. A thread is responsible for generating the following skip before it acquires the final skip from the list. To replace the node in the LoS, the thread needs to hold a node itself. This means that a node from the free list might need to be acquired (allocate).

P.2 A node can either be in the LoS, the free list, or be held by a thread.

P.3 A node can only be modified while it is owned by a thread.

P.4 A thread can hold at most one node at a time to ensure tight memory bounds. This means that the thread might need to free the node that it holds in order to acquire a skip from the free list (deallocate).

There are thus four ways in which a thread might go about acquiring a skip from the list of skips. One variable is whether the thread holds a node or not. The other variable is whether the list of skips only contains one node or more. How a thread interacts with the free list in these four situations is summarized in Table 1.

3.5 Example

To illustrate the exact operation of threads when acquiring and returning skips, we will explore a specific execution illustrated in Figure 4. In this example there are two threads T1 and T2 processing tuples that are to be sampled. In the figure, there are 3 nodes which are modified over time and each row corresponds to a point in time. Initially, at time (1), the LoS contains the left node, and the free list contains the two other nodes. The lists are designated by the lists' head pointers visualized by vertical arrows on the top pointing to their head. The nodes' successors are designated by the horizontal

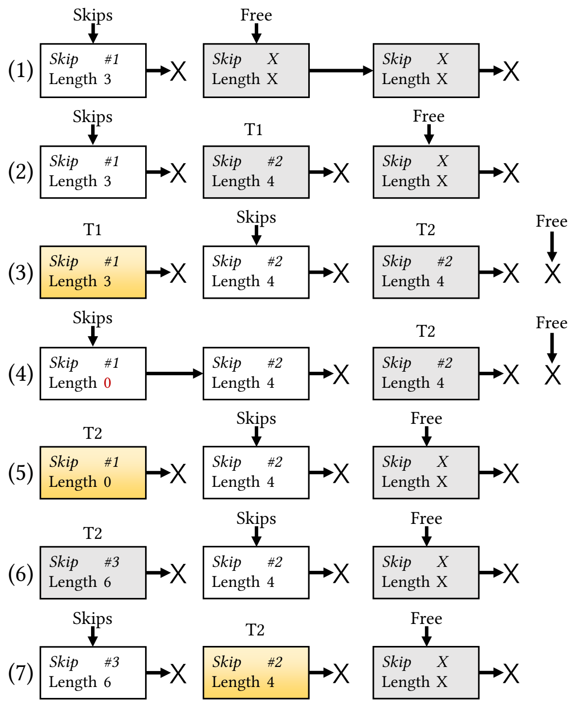

Figure 4: The LoS data structure

arrows pointing to the right. Nodes that point towards an X (*null*) have no successors, which means that they are the final nodes of their list. We will first go through the process from the perspective of T1. T1 starts at time (1). It reads the LoS' head pointer, and the head node on the left, checking whether it has a successor. As this is not the case, T1 first needs to generate the next skip in order to take the existing one as required by property **P.1**. T1 currently holds no node, so it must allocate one by taking a node from the free list as in Table 1 by moving the free lists head pointer to the successor of the free list head in a compare and swap loop. T1 succeeds and at time (2) it owns the middle node which it then fills with the information of the skip that has to follow skip #1 that T1 has seen at the head of the LoS. The information is generated by the Algorithm L as described earlier [13]. T1 also sets the successor pointer of its node to *null* as it will become the final node on the LoS. T1 then proceeds to set the head pointer of the LoS to the node it owns (middle node) with a compare and swap operation, ensuring that the head of the LoS has not been modified by another thread. After it succeeded at time (3), it now owns the left node with the first skip and the LoS now starts and ends with the middle node, which now contains the second skip. T1 then starts processing tuples. For every tuple it receives that is being inserted into the data set, T1 decrements the length of the skip by one if the length is greater than zero. T1 receives 3 tuples which it skips, decrementing its skip length to 0. Afterwards, it receives no further tuples and needs to shutdown, but it has not completed its range since it has not yet found a reservoir tuple. T1 must put its remaining skip back onto the LoS so that the corresponding range can be completed. So T1 sets the successor of the node it owns to the head of the LoS, and then sets the head pointer of the LoS to the node it owns with a compare and swap operation, ensuring that the head pointer has not been modified by another thread. T1 is now done.

The process is similar from the perspective of thread T2. It starts at time (2). Reading the LoS head pointer and the head node, it observes that the list only contains a single node. This leads T2 to allocate a node from the free list, and compute the skip following the LoS head node, storing the result in the node that it allocated. At time (4), T2 tries to replace the head node with the node that it holds containing the following skip, but fails since the head pointer has been modified by thread T1. Since the LoS contains two nodes, T2 first deallocates the node that it holds (right node) due to property **P.4**, and then proceeds to take the head node (left node). At time (5), the skip that T2 has acquired has skip length 0, meaning that the next incoming tuple is a reservoir tuple. After processing the next incoming tuple, T2 needs a new skip. At time (6), it checks the LoS, notices that the list only contains one node (middle node), then proceeds to compute the skip following the node in the LoS, storing the result in the node that it already holds (left node). At time (7), T2 replaces the head of the LoS with the node that it holds.

The final remaining case is where a thread holds no nodes and the LoS contains more than one node. In this case, no allocation or deallocation is necessary. The thread simply proceeds to move the head of the LoS forward to the successor of the head node and takes ownership of the prior head node.

3.6 Final Interface

Our sampling algorithm can be integrated into an existing system with a simple interface. The initialization of the sampling system requires allocation and initialization of the sample and the list of skips (LoS) data structure as described above. The sampling threads acquire skips at their initialization, run the skip processing logic for every tuple, and return their remaining skip on termination. Processing a skip in most cases only requires decrementing the thread local skip length, which results in almost zero overhead.

3.7 Preload

So far we assumed, that a sample already contains m elements. Initially, this is of course not the case. As long as the sample is not completely filled with m elements, special handling is required. The first m items passing through the input streams are guaranteed to end up in the sample. We take care of this in a special preload phase. In this phase, we keep track of the number of processed elements using a global atomic preload counter that is incremented for each item. The preload phase is finished, once this counter reaches m.

When a thread starts sampling, it first reads the value v of the preload counter. If $v \geq m$, the preload phase has already finished and the normal sampling procedure as described above can be applied. If the counter value is less than m, the preload phase is still in progress. In this case the thread performs an atomic fetch and increment instruction on the counter, atomically reading and incrementing its value. The newly read value v' can now again be larger than m if other threads have successfully processed further items in the meantime, thereby finishing the preload phase. In this case, the thread continues with the normal sampling procedure. Otherwise, preloading is still in progress and the thread tries to place the item into the sample at position v'. As the position might have already been written to by a thread performing the normal sampling procedure, the current thread will only write the element if the position is empty. For the next incoming item, the preload counter must be checked again until preloading is finished. Once the preload phase is finished, threads are not required to check the preload counter again.

3.8 Updates and Deletes

We have discussed how a single sample can be generated from streams of incoming data. In the context of databases, this corresponds to continuously inserting tuples into a relation. A full-fledged database additionally must support updates and deletes. Both of these result in changes to the original data set which may cause the sample to become outdated. Ideally, the sample should also reflect the resulting modifications. In the following, we therefore discuss, how our sampling strategy can be extended to support updates and deletes.

Olken and Rotem [18] describe an approach to support updates and deletes within samples. For updates, one needs to scan through the sample to find the affected tuple, and simply update the tuple in place if it is contained in the sample. Handling deletes is on the other hand slightly more involved: A deleted tuple, that is part of the sample, must be replaced by a different one. While Olken and Rotem pick a random tuple from the existing data set to replace the deleted tuple, our online approach needs to pick an upcoming tuple to replace the deleted tuple. After this replacement, the state of the sample will be as if no deletes actually happened, and the tuple from the future was actually inserted into the sample instead of the deleted tuple. To accomplish this, the operations on an upcoming tuple, must exactly replay the operations performed on the tuple being deleted. So we must first determine what actually happened to the original tuple. Was it placed into the reservoir, or was it skipped? If the deleted tuple was skipped, we increment a skip length of a skip node by 1. This will result in an extra tuple being skipped in the future which will take on the role of the deleted tuple. If the deleted tuple was placed into the reservoir, a new skip node with a skip length of 0 is inserted to the front of the LoS. Due to this new skip of length 0, an additional upcoming tuple will be placed in the sample thereby replacing the deleted one. It must, however, be ensured, that the newly inserted tuple will end up at the exact same position in the sample as the deleted one. This is achieved by using a deterministic counter based random number generator [19] with the skip index as the counter, which roughly corresponds to hashing the skip index, to generate the random positions in the sample. Thus, by giving the newly inserted skip node the skip index of the deleted tuple, the newly inserted tuple is guaranteed to overwrite the deleted one.

The most expensive part of this approach is to scan the reservoir to check whether it contains a certain tuple.This can be greatly optimized by storing whether a tuple has been inserted into the reservoir within the data set itself while inserting the tuple. Thereby, whether a tuple is part of the reservoir can be checked during updates and deletes by reading the original data set, which has to be touched anyway. Only if the tuple is part of the reservoir, the sample has to be scanned for the tuple. While this causes a minimum storage overhead with one bit per tuple, it still requires scanning the sample, although exponentially less often. Scanning the sample for a tuple can also be completely avoided at the cost of slightly higher storage overhead. Since the position of a tuple in the sample can be recomputed based on the skip index, storing the skip index of reservoir tuples in the dataset allows to directly access sample tuples without having to scan the entire sample.

4 ANALYSIS

We shall now analyze our algorithm's results and efficiency. Our approach implicitly orders the tuples concurrently processed by different threads. When a thread acquires a skip, it skips a number of tuples and determines a tuple to be a reservoir tuple. The order of the tuples that are processed as part of the same skip is their processing order. The order of tuples processed as part of different skips is the order of the corresponding skip indices. This ordering essentially serializes every concurrent stream into a single stream of data, thus implicitly assigning an index to every tuple within a global ordering of the entire set of tuples. If we were to compute a sample with the single-threaded algorithm due to Li [13] with the serialized data set, we would construct the same sample as our multi-threaded algorithm, simply because the same decisions would be made for the same tuples.

This explanation makes the simplifying assumption that there is no index that has not been assigned. Consider the case when two threads are processing two skips, but neither of them has completed theirs, i.e. has found a reservoir tuple. There are two non-contiguous ranges of indices that have been assigned to tuples, and in between, there is a hole where indices have not been assigned tuples. We cannot find an ordering of the tuples for the single-threaded algorithm that would have the same results as the processing by these two threads. If this hole were to be filled, we would have one additional reservoir tuple, and the sample would potentially differ by a single element. Such missing reservoir tuples are bounded by the number of threads, since for every thread, there is at most one skip that is not completed. And missing reservoir tuples with relatively small orders have a high probability of being replaced by later reservoir tuples, so the importance of determining them decreases with increased amount of tuples processed.

Accepting a limited amount of missing reservoir tuples allows our approach to have low communication costs. The number of skip acquisitions is expected to be $O(m \cdot (1 + log(n/m)))$, where m is the size of the sample and n is the size of the data set being sampled. This complexity is the expected number of skips needed for sampling the data [22]. It includes the $O(m)$ operations required to initially fill the sample. Additionally, there are a minimum of $O(t)$ skip acquisitions, one for each of the t threads. This results in a total of $O(m \cdot (1 + log(n/m)) + t)$ required shared memory operations, which is less than the lower bound of $\Omega\left(\frac{t}{log(1+t/m)} \cdot log\left(\frac{n}{m}\right)\right)$ for sampling algorithms that require the sample to represent the entire data set at all times [21].

The use of compare and swap (CAS) instructions allows for efficient synchronization. As described in the Section 3, in most cases, a single compare and swap is required to acquire a skip. A second compare and swap is only required when interacting with the free list, which is not required in cases where already initialized

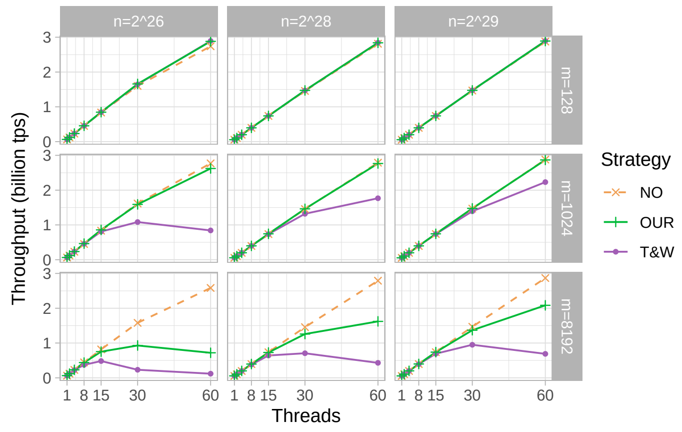

Figure 5: Throughput in microbenchmark

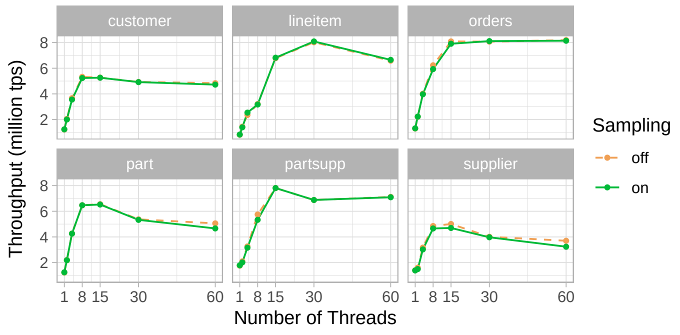

Figure 6: TPC-H bulk-loading throughput in Umbra with and without sampling

threads are inserting many tuples. Those threads will simply reuse the nodes they already own, without touching the free list.

CAS operations can fail when multiple threads are running. However, if a thread fails to acquire a skip, another thread must have succeeded, which guarantees forward progress. Additionally, a thread that fails is more likely to acquire the next skip, since the successful thread needs to process the skip it has acquired. In practice, this avoids thread starvation even in highly concurrent setups.

Since the sample is updated at random positions, there is almost no contention between threads. In the approach due to Tirthapura and Woodruff [21] random variates for tuples in the sample are kept in an ordered list. Maintaining an ordered list potentially requires $O(log(m))$ operations per update. Our approach's operations on shared memory in contrast are designed to be concise and efficient.

5 EVALUATION

We have evaluated our approach on a 4 socket Intel(R) Xeon(R) E7-4870 v2 @2.30GHz machine with 15 cores per node and 60 cores in total. The system is a NUMAcc (non-uniform memory access with cache coherency) system. This means that from the application's point of view, it acts like a single socket machine in terms of cache behavior. However, depending on the node to which accessed memory is attached to, memory access latencies vary.

5.1 Microbenchmark

We first evaluate our sampling strategies performance in processing large data sets using an isolated microbenchmark. The benchmark measures throughput in tuples per second (tps) with various settings for thread count, sample size and size of the data set. Tuples are generated by each worker as strings ranging from 3 to 11 bytes based on the index of the tuple. Threads receive similar amounts of tuples, but many threads receive tuples of larger sizes, while a few threads receive many small tuples to simulate a heterogeneous workload. However, the general size of the tuples is kept small to focus on the overhead of sampling. Tuples are copied into the sample, if they were decided to be put into the sample.

We build samples of size 128, 1024 and 8192 tuples from data sets of size 2^{26}, 2^{28}, and 2^{29} tuples. Tuples are evenly distributed among worker threads. We measure the total time taken to sample these data sets and compute the throughput by dividing the total size of the data set by the time it takes to process it. We report the average of 10 repeats of each experiment in Figure 5. We compare two strategies and a baseline strategy designated as NO for *no sampling*. NO performs no sampling, it simply iterates through the tuples in parallel with no communication between threads. The *OUR* strategy is our strategy that uses the list of skips (LoS) data structure to maintain low memory bounds. The *T&W* strategy is the strategy due to Tirthapura and Woodruff [21]. Since no explicit shared-memory strategy is described in their work, we use a shared atomic integer to communicate the largest variate in the sample. If the sample needs to be modified, a spin-lock is used to synchronize access to the ordered list of variates, which we implemented as a fast binary heap. We measure throughput in tuples per second by dividing the total number of tuples in the data set by the total elapsed time in seconds. An optimal algorithm is expected to scale linearly.

Our approach scales virtually perfectly for sample sizes 128 and 1024, even across 60 threads on four NUMA nodes. Both the low amount of communication, and the simplicity of the synchronized list of skips operations result in impressive scalability. With large sample sizes, the necessary communication is much higher since many more tuples are put into the sample. Even in those cases, our algorithm can still benefit from higher numbers of threads, especially as the data set grows and the frequency of sampled tuples decreases. Note that, if contention is high, more threads could even result in drastically lower performance. Even in the worst case with a smaller data set, the largest sample, and 60 threads, our algorithm still maintains comparatively high throughput.

5.2 Integration in Umbra

We also integrated our sampling strategy into our high performance, fully SQL query capable, database system Umbra [17], which previously used to recompute samples of base relations periodically, incurring massive random I/O costs. In Figure 1, it can be seen that sampling 1000 random tuples may take longer than a minute in large data sets with cold cache. This is unacceptable when answering queries that may themselves execute in less than a second. We evaluate the overhead of our sampling strategy to the insert operation in Umbra in bulk-loading TPC-H relations into the database where those relations are sampled into a sample of size 1000. Results are shown in Figure 6. We vary the number of threads from 1 up to 60 and can see virtually no difference in insert throughput with sampling on or off. As in any other database, the insert operation itself requires a lot of synchronization and computation, such that the overhead of sampling is in comparison, essentially unnoticeable. This supports our claim that our sampling strategy has almost zero overhead for many workloads that require sampling.

5.3 Empirical Evaluation of Sample Quality

While we have a theoretical approach to evaluate sample quality, we are not aware of an empirical measure to test the quality of a given set of generated samples. Statistical tests might potentially be useful to check whether generated samples are unlikely to be generated by a correct sampling strategy. However, we are not aware of a widely accepted test for sampling without replacement. If we are to assume that all the elements within the sample are independently and identically distributed with the uniform distribution, which can generally be assumed for data sets that are much larger than the samples, one could apply the Anderson-Darling test [2] to a given sample. To apply the Anderson-Darling test, we first determine the critical value. For a significance level of 5%, based on 100,000 samples of 1024 uniformly distributed random numbers, we calculate this critical value as 2.48884. Using this critical value, we run the Anderson-Darling test on the samples of size 1024 generated by our algorithm in the microbenchmark with the data set of size 10^{29}. Around 96% of the samples generated by our algorithm pass the test, which is in accordance with the significance level of 5%. Note that this is more of a sanity check rather than a proof of sample quality. These results just show that it is not trivial to distinguish a sample generated by our algorithm and a sample generated by a traditional sampling algorithm.

6 CONCLUSIONS & FUTURE WORK

We have presented an efficient, highly scalable, and concurrent approach to online sampling. While the complexity of algorithms in prior work always depends on the number of threads, thus impeding scalability, the algorithm we propose scales perfectly, as its complexity is independent of the degree of parallelism. We achieve this scalability by temporarily relaxing the requirements on a sample's freshness. A theoretical analysis as well as an experimental evaluation shows both the minor impact of those relaxations on sample quality and the superiority of our algorithm's runtime performance.

ACKNOWLEDGEMENT

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] Mohammed Al-Kateb, Byung Suk Lee, and X. Sean Wang. 2007. Adaptive-size reservoir sampling over data streams. In *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*. https://doi.org/10.1109/SSDBM.2007.29

[2] T. W. Anderson and D. A. Darling. 1954. A Test of Goodness of Fit. *J. Amer. Statist. Assoc.* 49, 268 (1954), 765–769. https://doi.org/10.1080/01621459.1954.10501232

[3] Altan Birler. 2019. Scalable Reservoir Sampling on Many-Core CPUs. In *Proceedings of the 2019 International Conference on Management of Data - SIGMOD '19*. ACM Press, New York, New York, USA, 1817–1819. https://doi.org/10.1145/3299869.3300096

[4] Yung Yu Chung, Srikanta Tirthapura, and David P. Woodruff. 2016. A Simple Message-Optimal Algorithm for Random Sampling from a Distributed Stream. *IEEE Transactions on Knowledge and Data Engineering* (2016). https://doi.org/10.1109/TKDE.2016.2518679

[5] Graham Cormode, S. Muthukrishnan, Ke Yi, and Qin Zhang. 2010. Optimal sampling from distributed streams. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. https://doi.org/10.1145/1807085.1807099

[6] Graham Cormode, S. Muthukrishnan, Ke Yi, and Qin Zhang. 2012. Continuous sampling from distributed streams. *J. ACM* (2012). https://doi.org/10.1145/2160158.2160163

[7] Maurice. Herlihy and Nir Shavit. 2012. *The art of multiprocessor programming* (revised fi ed.). Morgan Kaufmann. 508 pages.

[8] Lorenz Hübschle-Schneider and Peter Sanders. 2019. Communication-Efficient (Weighted) Reservoir Sampling from Fully Distributed Data Streams. (oct 2019). arXiv:1910.11069 https://arxiv.org/abs/1910.11069

[9] IBM. 1983. IBM System/370 Extended Architecture, Principles of Operation, publication no. SA22-7085-0. (1983), A–44.

[10] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings - International Conference on Data Engineering*. https://doi.org/10.1109/ICDE.2011.5767867

[11] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right : Index-Based Join Sampling. In *Cidr*. http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf

[12] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. In *VLDB Journal*, Vol. 27. 643–668. https://doi.org/10.1007/s00778-017-0480-7

[13] Kim Hung Li. 1994. Reservoir-Sampling Algorithms of Time Complexity O(n(1 + log(N/n))). *ACM Transactions on Mathematical Software (TOMS)* 20, 4 (dec 1994), 481–493. https://doi.org/10.1145/198429.198435

[14] Guido Moerkotte and Axel Hertzschuch. 2020. alpha to omega: the G(r)eek Alphabet of Sampling. In *{CIDR} 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p25-moerkotte-cidr20.pdf

[15] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993. https://doi.org/10.14778/1687627.1687738

[16] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved selectivity estimation by combining knowledge from sampling and synopses. In *Proceedings of the VLDB Endowment*, Vol. 11. 1016–1028. https://doi.org/10.14778/3213880.3213882

[17] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. *Cidr* (2020).

[18] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5, 1 (1995), 25–42. https://doi.org/10.1007/BF00140664

[19] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. 2011. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/2063384.2063405

[20] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. 2016. Efficient Random Sampling - Parallel, Vectorized, Cache-Efficient, and Online. http://arxiv.org/abs/1610.05141

[21] Srikanta Tirthapura and David P. Woodruff. 2019. Optimal Random Sampling from Distributed Streams Revisited. (mar 2019). arXiv:1903.12065 http://arxiv.org/abs/1903.12065

[22] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57. https://doi.org/10.1145/3147.3165

[23] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüş, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *Proceedings - International Conference on Data Engineering*. https://doi.org/10.1109/ICDE.2013.6544899