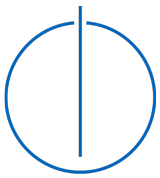
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Maintaining a Sample Under Concurrent Updates

Michael Zinsmeister



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Maintaining a Sample Under Concurrent Updates

Sampleverwaltung unter parallelen Änderungen

Author:	Michael Zinsmeister
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Altan Birler
Submission Date:	15.04.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.04.2024 Michael Zinsmeister

Acknowledgments

First and foremost I want to thank **my family** who helped, guided, and always supported me throughout my almost 20 years of school and university up to this point where I have the privilege to graduate with a master's degree from TUM.

Additionally, I want to thank my advisor **Altan Birler** who provided invaluable feedback on my work and helped me find my way around the Umbra code base. I also want to thank my supervisor **Prof. Dr. Thomas Neumann** for giving me the opportunity to work on this topic and do so within Umbra. Special thanks should also go to **Professor Leis** whose courses, in addition to Professor Neumann's, are responsible for most of my database systems knowledge and skills.

Finally, I'd like to express my gratitude to several dear friends and colleagues who have been integral to my journey through academia and professional life. Firstly, I extend my thanks to **Rafael**, a lifelong friend whose unique perspectives have enriched my life since our days in kindergarten. I also want to acknowledge **Hans**, whose remarkable intellect has been a constant source of inspiration during our years together at school and university, fueled by our shared passion for computer science. Both **Tobi**, my companion since day one at HM, and **Alex**, my companion in most database courses at TUM, made my time at university much more enjoyable. Furthermore, I am appreciative of the valuable experience gained from working with **my colleagues at NOVENTI** as a working student for over 5 years. Special acknowledgment goes to **Peter and Andy**, whose guidance and mentorship significantly contributed to my growth as a software engineer. I am also thankful to my counterparts in the management team at TUMuchData – **Georg, Marlene, Paul, Sebastian, Guowen, Pascal, and Christoph** – for enriching my time at TUM with their camaraderie over the past months.

Abstract

Having accurate estimates for certain statistical properties available is one of the most important factors for getting good results from query optimization in database systems. To always have up-to-date statistics available, it is preferable to maintain them online instead of periodically recalculating them in the background. A sample is a good way to estimate the selectivity of filter predicates. Therefore, it is desirable to maintain an online sample.

The Umbra database system has an existing implementation of such online statistics-maintenance including an online sample. This implementation is performant and scalable but does not support update or delete operations. Many real-world workloads can significantly degrade the quality of a sample if updates and deletes are not supported.

We show that deletion can be achieved at minimal performance cost by using a carefully designed concurrent hash table for finding the position of tuples in the sample and a scalable implementation of the random-pairing method for implementing delete operations on top of arbitrary reservoir sampling algorithms. Updates can also be handled using the previously mentioned hash table. Benchmarks show that an implementation of this in Umbra can achieve TPC-C results in the hundreds of thousands of transactions per second just like unmodified Umbra with less than 5% of performance degradation.

Umbra uses a compressed representation of the online sample for faster cardinality estimation. This cannot be updated online, though, and therefore has to be recompressed after significant changes in the online sample. To determine when this is necessary, a method for the estimation of set-difference between the online sample and the compressed sample is needed. Our implementation using OddSketches and a combined hash of all columns can be used to accomplish this with high accuracy.

Contents

1. Introduction

Query optimization is one of the most important steps in query processing for the performance of a database system. Many query optimization algorithms depend on estimates derived from statistics. It has been shown that the accuracy of estimates is one of the most important factors that determine the quality of optimization results and that industrial-strength cardinality estimators often produce large estimation errors resulting in suboptimal plans being generated [Lei+15]. Even if an optimizer produces theoretically optimal plans, this optimality only applies under the assumption that estimates are perfect. Therefore, increasing the accuracy of estimates through better statistics can be very beneficial. The freshness of statistics is a major factor for this. Ideally, statistics should always represent the current data and not be out of date. This can be achieved by maintaining them online as the data is modified. This kind of statistics maintenance can only achieve good results, however, if it supports all insert, update, and delete operations.

Most well-known commercial and open-source systems mainly use histograms for modeling the data distributions of single columns. Those histograms are then rebuilt regularly or after the data changes significantly. Random sampling is an alternative to this. It has the benefit of representing the data and correlations between different attributes rather accurately. Probabilistic data structures called sketches can also be used to keep estimates of certain statistical properties. For example, HyperLogLog sketches can estimate distinct value counts.

1.1. The staleness problem

Staleness of statistics is a major issue in traditional database systems. Bulk operations can, for example, change the statistical properties of a database table significantly within a short amount of time. If a system only updates statistics periodically, those changed properties will not be represented in statistics in any way until the next recalculation. Even if the recalculation is started immediately after a bulk operation, it will take time to execute this for large datasets.

Therefore, Umbra [NF20] uses sketches and a random sample [BRN20], which are all maintained in an online fashion, for statistics. This means that as new tuples are inserted into a relation, the sample and sketches are updated live to always be an

accurate representation of the data. However, the sample is only kept up to date for transactions that insert data into the relation, not for other modifications like updates or deletes. Bulk deletes or bulk updates can have the same negative impact on statistics quality as bulk inserts if they are not applied to the statistics. Therefore, statistics in a database system should support online maintenance for insert, update, and delete operations. This should, however, not negatively impact the efficiency of these operations. For sketches, working solutions for this already exist, e.g. described in [FN19] for HyperLogLog sketches. The goal of this thesis therefore is to show ways in which a sample can be kept up to date with the live data at all times and do so with only minimal performance impact. This will be achieved by utilizing a concurrent hash table for finding tuples in the sample and a sharded implementation of random pairing [GLH06] with a thread-local state for bulk operations.

Umbra uses a compressed representation of the sample for estimating cardinalities more quickly. The compressed sample is static and needs to be recomputed from the online sample after the data has changed significantly. This can be done in rather little time but should still not happen too often. Hence another challenge is finding a way of quickly estimating the difference between the compressed sample and the online sample to decide when the compressed representation should be recalculated from the online sample. This will be achieved using OddSkeches [MPP14] with hashes that can quickly be recalculated without rehashing an entire row even for single-column changes.

1.2. Motivating example

As a motivating example for why update and delete are important in an online sample, a generic trading company can be used. This company has a database for its order and delivery workflow. Each order has a status attribute. This status can either be *NEW*, *PROCESSING*, *DELIVERING* or *DONE*. New orders are inserted with status *NEW*. The vast majority of the orders will usually be *DONE* if orders are never deleted.

If one only updates the statistics for inserts into the orders database table, however, the statistics will tell that the status is always *NEW*. Therefore, the company might be wondering why their analytics queries for new orders are slow. It might turn out that the database system thinks that it has to look at almost all rows inside that table anyway and therefore will choose to scan the entire table instead of using an available index on the status. This would be the right choice if the statistics were accurate. Since they are not, however, the database might have to read millions of rows instead of doing an index scan with just a few hundred rows. This kind of pattern appears often in real-world workloads.

1.3. Outline

The overall structure of the rest of this thesis is as follows: Chapter 2 is about relevant theoretical background and related work used in later chapters. Afterward, to add the necessary context for the implementation part, Chapter 3 explains and analyzes the currently implemented but unpublished scalable reservoir sampling method implemented in current mainline Umbra. The main part of the thesis follows in Chapter 4. It proposes several implementations of random pairing, random replacement, and set difference estimation. Additionally, it provides some implementation considerations for supplementary areas. Afterward, the solutions are compared using several benchmarks in Chapter 5 which is followed by a short discussion of the results this thesis produced in Chapter 6.

1.4. Contributions

The main contributions of the thesis are:

- A scalable implementation of random pairing
- An evaluation of different random-pairing implementations
- An evaluation of using random replacement and methods for implementing it
- A method for quickly estimating the similarity between two samples with minimal memory usage

2. Background

The implementation described later will build up on some theoretical basics which will be explained in this chapter. Random Sampling and Reservoir Sampling explained in Section 2.1 are fundamental to online sampling in database systems. In Sections 2.1.2 and 2.1.3, we explain Tirthapura and Woodruff's algorithm and Li's Algorithm L which are two specific reservoir sampling algorithms that Umbra's sampling logic builds upon. Section 2.2 describes Random pairing. It is the basic method for supporting deletes in reservoir sampling, for which this thesis describes a performant and scalable implementation. Another option for handling deletes is random replacement (Section 2.3), for which we need to also be able to sample from a B+Tree with uniform probability. Some probabilistic data structures used for maintaining further statistics and for estimating the difference between two samples are described in Sections 2.4 to 2.8. Lastly, an overview of Umbra's architecture is given in Section refsec:umbra to have the necessary context to understand the implementation described in Chapter 4.

2.1. Random Sampling

Random Sampling describes the selection of a random sample from a base dataset such that this sample is representative of the base data. In this thesis, *random sample* or just *sample* always describes a *simple random sample*. This means that every subset of size k in the base dataset is as likely to be in the sample as any other subset of the same size. A sample is also always assumed to be without replacement from here on so that it contains no duplicates.

Since sampling is not limited to database use cases, the term *data point* will be used for the possible elements of the sample in this chapter's general description of sampling. A data point can be arbitrary data and is usually treated as a black box by the sampling algorithm. Tuples in a database are one example of data points. Later chapters explaining the implementation will therefore talk about tuples in a database specifically instead of generic data points.

There are a multitude of possible applications for random sampling. Most of them are about quickly getting an approximate aggregate value for some subset of a dataset where the subset is significantly larger than $\frac{|dataset|}{|sample|}$. This is often used for stream pro-

cessing, which usually means processing a large number of data points and answering queries about them without having to store all the elements.

A random sample can be used for approximating all kinds of statistics in very little time compared to exact processing on the base dataset, which is assumed to be orders of magnitude larger. One example of such a use case is approximating selectivities of filter predicates in database systems for query optimization.

2.1.1. Reservoir Sampling

Reservoir Sampling describes a general type of sampling strategy that allows generating a random sample with a single pass over the data and without knowing the data size before processing it. An additional attribute of reservoir sampling is that it always maintains a representative sample of the already processed data during this process. Many different algorithms with different properties can achieve this. All of these algorithms can be categorized as reservoir sampling algorithms [Vit85]. Such an algorithm decides, for each (new) data point, whether to insert it into the sample and replace another sample element with it.

Delete operations are usually not handled by reservoir sampling algorithms. Just deleting a data point from the base data and the sample makes the remaining set still a valid sample of a smaller size. When handling new inserts afterward, however, some special logic needs to be applied. Otherwise, deleting and reinserting the same data over and over again would lead to the amount of data points skipped before one is inserted into the sample afterward being much larger than without the delete-reinsert sequence. Therefore, newer data points would be underrepresented in the sample.

Birler et al. have proposed a parallel online reservoir sampling algorithm based on a *list of skips* approach [BRN20]. The *list of skips* essentially is a fixed-size lock-free linked list. This approach also includes a general idea of how to handle deletes, which has two major limitations: Firstly, it could lead to the list of skips overflowing, which would need some special care. Secondly, it requires setting a maximum number of inserting threads before use. Later chapters will therefore describe and propose a different approach.

2.1.2. Tirthapura and Woodruff Distributed Streams Reservoir Sampling algorithm

Umbra stopped using the sampling method described in [BRN20] in version d2049f39 (January 2023) and is using a different reservoir sampling method based partly on one originally proposed for use in distributed systems and partly on Algorithm L described by Li [Li94].

Tirthapura and Woodruff describe this distributed reservoir sampling algorithm [TW11] that assumes one master node that manages the sample and a number of workers that process an incoming stream. This algorithm also has some desirable characteristics for a multi-threaded single-node environment because it also has reduced communication between threads and therefore contention. It is used in a form adapted for a single node but multi-threaded case in Umbra for maintaining an online sample and will therefore be used as a base for the implementation described in this thesis.

It works by generating a random number for each incoming data point. The data points having the n smallest random numbers are then in the sample, similar to [Li94]. Therefore the algorithm needs to keep track of the nth lowest number seen so far, which is also the highest number whose corresponding data point is in the sample. This serves as a threshold for including a data point into the sample and replacing another one. A new data point is only taken into the sample if its random number is smaller than the nth lowest number. The original paper does not go into detail on how to maintain this number. A simple but unscalable way to implement this is a synchronized max-heap.

To avoid having to communicate updates to the threshold with all nodes, each node only ever knows the threshold shortly after it last sent a data point for inclusion in the sample. Whenever it gets a data point with a random number smaller than this threshold, it sends it to the master node and gets back the new nth lowest number. The master then makes the final decision of whether the new data point will be inserted into the sample or not according to the actual current threshold. This of course leads to nodes tending to have slightly too high thresholds and therefore sending slightly too many points to the master. Since the master will, however, check the random number of the new data against the actual current threshold this is corrected again.

2.1.3. Reservoir sampling Algorithm L

As mentioned in the previous section, Umbra does not use a pure Implementation of Tirthapura and Woodruff but instead combines it with ideas from a different algorithm. Li describes multiple reservoir sampling algorithms [Li94]. Algorithm L is the one Umbra borrows some ideas from.

Its basic idea is also generating random numbers for each data point and only keeping the data points with the n smallest numbers. However, instead of actually generating a random number for each data point, Algorithm L samples from a geometric distribution to pre-calculate a number of incoming data points to skip before a new one is taken into the sample. Also, instead of maintaining the exact threshold as required in Tirthapura and Woodruff's algorithm, Algorithm L just samples this from a distribution too whenever a new data point is inserted into the sample. The distribution used for this is one describing the largest value in n uniform random values between 0 and the old

threshold.

2.2. Random Pairing

Random pairing as described by Gemulla, Lehner, and Haas [GLH06] is one theoretical option for including deletes in a random sample maintained by some reservoir sampling algorithm. It works without accessing the base data after the initial insertion. This means that it is equally suitable for both stream processing and database systems use cases. It will therefore be used as the basic method for most implementations described in Chapter 4.

2.2.1. Timeline model of sample maintenance

To reason about sampling, it makes sense to reason about operations to a sample on a timeline. This timeline shows all points in time at which a data point was considered for inclusion into the sample. At each such point, the reservoir sampling logic makes a decision on whether to include or ignore the data point for the sample. Now deletes and replacements for these deletes can be considered in this context. Figure 2.1 shows a basic visualization of this model. Shown are inserts on a timeline with the sampling decision color-coded.

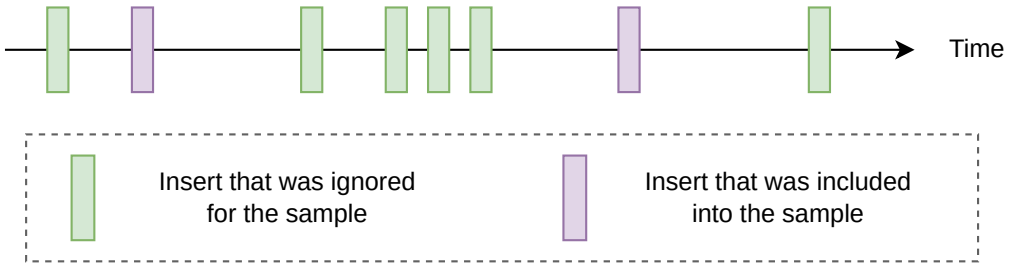


Figure 2.1.: Basic visualization of the timeline model with inserts

2.2.2. Requirements for a delete handling strategy

The most important requirement for a delete handling strategy in a sampling context is that the statistical properties of the sample still hold after all deleted data points in the sample have been replaced by new data points. Therefore, just continuing with the sampling process will not be good enough. The deleted data points have to be handled such that the sample looks like they were never considered for inclusion in the first place. Only then will a sample still be representative after they were replaced

with new data points. Otherwise, the amount of ignored data points between two included data points would be too large. In other words, newer data points would be underrepresented. In the extreme case where all data points that have ever been inserted into the sample are deleted, the sampling algorithm would ideally have to be reset to the state it had when it was first initialized. This, however, only works for this one extreme case and not for any other case. Thus, the sample will usually not be full again until almost the same number of data points that have previously been deleted are inserted again. Otherwise, the newer data points after the deletes would be overrepresented in the sample.

The only way to have the sample filled again faster would be to sample from the base data which is assumed to not be accessible after the insert for the random-pairing strategy. This is a valid strategy because adding a random data point to a valid random sample will create a new valid random sample as long as the new data point is not already part of the sample, which would create duplicates. A new random sample can also be created by just starting with an empty sample and repeating this sampling step until the sample has the desired size. In case of reservoir sampling, the sampling algorithm would then, however, need to be reset using the reduced size of the base dataset. This would then be equivalent to a situation in which the deleted data point was never inserted in the first place.

Figure 2.2 shows this in the timeline model. Even after they are deleted it is still important whether the deleted data points had previously been skipped or included into the sample to be able to replace them accordingly.

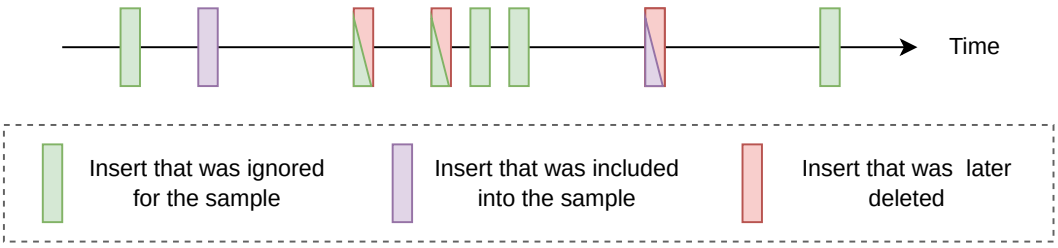


Figure 2.2.: Deletes in the Timeline model

2.2.3. Finding replacements through random pairing

The random pairing strategy [GLH06] suggests finding a random "partner" for each deleted data point in the stream of newly inserted data points before continuing with the normal sampling logic. This can be visualized on the timeline by pretending the new inserts travel back in time to the point where some randomly selected previously

deleted data point was inserted. The decision for whether this data point was included into the sample has also already been made at an earlier time and can just be copied from this previous insert.

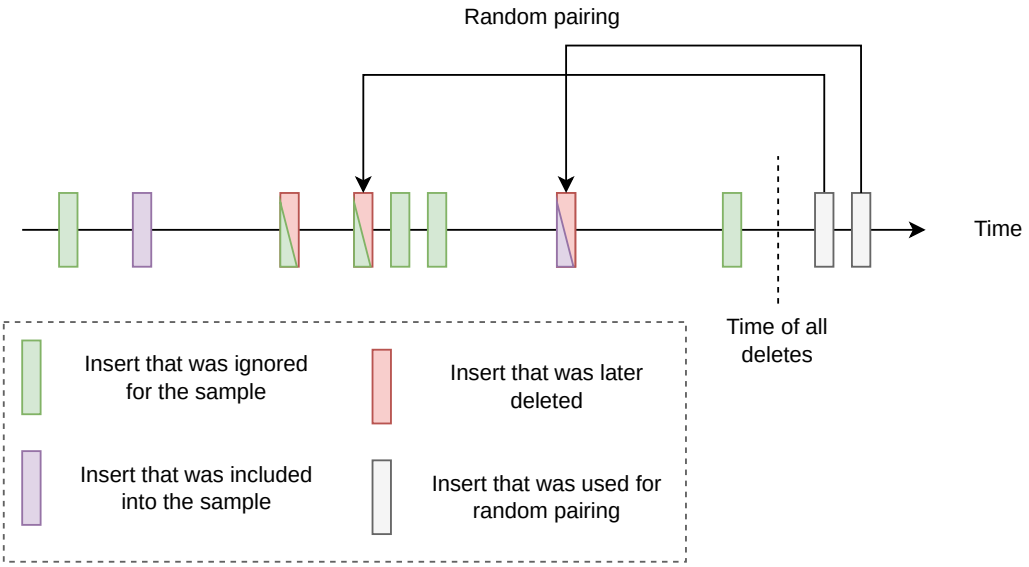


Figure 2.3.: Random pairing in the timeline model. New inserts travel back in time.

In Figure 2.3 an example of this thought model is shown. First, some data points are inserted using normal reservoir sampling (e.g. using the method described in Chapter 3). At a certain point in time, three data points are deleted of which one was a part of the sample. After the deletes are done two data points are inserted which will then randomly be paired with the previous deletes. In this example, the first insert will be paired with the second delete and not be placed into the sample since its partner also was not in the sample. The second insert, however, is paired with the third insert whose data point was a part of the sample. Therefore, the second insert is placed into the sample. For the sake of simplicity, it can be assumed that none of the data points that were placed into the sample during the depicted part of the timeline replaced any other data points placed into the sample during that time frame.

This is just a thought model, however. In reality, it is enough to maintain two counters. One counter (c_1) will be used for tracking the number of data points deleted that were in the sample. The other counter (c_2) will be used for tracking the number of data points deleted that were not in the sample. Those will need to be compensated by newly inserted data points too.

Then, for each newly inserted data point, a decision strategy will be picked depending

on whether the counters are both 0 or not. If they are both 0 the regular sampling strategy will be used. If one of them is not equal to 0, then the random-pairing strategy will be used. To decide whether a data point will be taken into the sample a random number between zero and one will be generated and if that number is below $\frac{c_1}{c_1+c_2}$ the data point will be included in the sample at the position of a previously deleted data point [GLH06]. After deciding whether to take the data point into the sample or not the respective counter will be decreased. This can also be optimized further by sampling a number of data points to skip instead of deciding on a per-data-point basis.

2.3. Random replacement

An alternative handling strategy for deletes, which will also be evaluated later, is to sample replacement data points for deleted ones randomly from base data. This approach is only possible if base data access is an option. It can result in significant cost if base data access is expensive and more importantly, it is also oftentimes not trivial to sample a data point from an actual base data store in an unbiased way. Especially with variable-sized data, even sampling from heap-pages becomes nontrivial. This is because it will often only be possible to sample byte positions or pages uniform-randomly. For byte positions, a twice as-large data point will have twice the probability of being sampled. For pages, twice as large data points on full pages will also have twice the probability of being sampled. Umbra has a B+Tree as its main storage data structure for relations. Therefore, a method for uniformly sampling from those B+Trees is required to get unbiased random replacements.

2.3.1. Acceptance/Rejection Sampling

As previously explained, sampling from base data is often not trivially possible. Therefore a general method of correcting for the biases of B+Trees or variable-sized data is needed. Acceptance/Rejection Sampling makes this possible. It is used to transform a random variable from a uniform probability distribution into an arbitrary other probability distribution using just the Probability density function (PDF) of the target distribution. This means it enables sampling from any probability distribution, even ones whose PDF cannot easily be integrated to get a cumulative density function that could then be used to transform random variables from one distribution to another.

Acceptance/Rejection sampling works by first sampling a value from the space of available values and then sampling another number between 0 and the global maximum of the PDF or an upper bound approximating it. If this number is then less or equal to the value of the probability distribution function at the first value, a valid random value

has been sampled from the target distribution. If it is higher, the sampling process has to be restarted.

The higher the ratio between the expected value of the PDF and its global maximum, the higher the average number of restarts will be and the higher the cost for sampling a single random value will be.

2.3.2. Random sampling from B+Trees

To achieve truly random sampling from B+Trees, it is not enough to just do a random walk through the tree to a leaf and randomly pick one of its values. This is because nodes can have a fanout of anywhere between f_{max} and $\frac{f_{max}}{2}$ where f_{max} is the maximum fanout of an inner page. Hence, the probability of a child being picked can also be anywhere between $\frac{1}{f_{max}}$ and $\frac{2}{f_{max}}$.

The argument could be made that over time, assuming a constant influx of inserts and deletes, those numbers will change and therefore still produce an approximate uniform distribution. However, especially for a B+Tree like the one used by Umbra to store tuples by Tuple ID (TID), this might not be the case since deletes might have patterns and TIDs are not reused. There is a theoretically correct way to achieve a truly uniform distribution, though. Ranked B+Trees [OR89] could achieve this, however, they are expensive to maintain and only useful for read-only or read-mostly workloads. But even using just regular un-annotated B+Trees this can be accomplished by using acceptance/rejection sapling [OR89]. What this effectively means is pretending to walk down a tree of full nodes with maximum fanout and restarting the sampling whenever a position that is larger than the number of actual child nodes is drawn. This can lead to quite some overhead but the original authors of the paper describing this approach determined it would only be worse than sampling from an annotated tree by a factor of about three for a B+Tree of depth seven. The original paper assumes fixed-size keys and values. This problem can be solved in a similar way, which will be described in Chapter 4, though.

2.4. Probabilistic data structures

Some probabilistic data structures are used by Umbra and some additional ones will be used in the implementation in Chapter 4. Probabilistic data structures are data structures that rely on some pseudo-random element, usually hashing, and some of its statistical properties to get approximate answers to different kinds of questions. In database systems, they can be used either in a user-facing way for quick approximate query processing or internally for statistics maintenance.

2.5. HyperLogLog sketches

The HyperLogLog sketches are a probabilistic data structure used in Umbra for counting the number of distinct values in a column. They form a different kind of statistic also maintained online. For some of the further work in this thesis, it will make sense to consider statistics as a whole, including HyperLogLog sketches.

They utilize the fact that getting a series of n leading zeros in a uniformly distributed random binary number has probability 2^{-n}. Hash values generated by good hash functions like MurmurHash3 [App] or Wyhash [Yi] can be considered random in the sense that they will usually map almost any set of inputs to a set of numbers that will also be uniformly distributed over the entire space of output values, ideally without any patterns. HyperLogLog sketches therefore keep track of the highest number of such leading zeros that was encountered in the hashes of the values to be counted [Fla+07]. Using the property described above, the number of distinct values can then be estimated through the expected number of distinct random values required to get one with the observed maximum leading-zero count. This can be further improved by not only keeping track of one value but first dividing the hashes into p buckets, or *substreams* as they are called in the original paper, and then keeping track of the number of leading zeros for all buckets. This minimizes the effect of outliers through statistical averaging [HNH13].

Since the values present in a single column of a database table form a multiset and will also often actually have many duplicates, HyperLogLog sketches can be used to keep an accurate estimate of how many distinct values a column has. This is helpful for join size estimation.

2.6. Bloom Filters

Bloom Filters are a probabilistic data structure used to quickly test set inclusion. They are used to optimize the hash table described in Section 4.1 for the case of misses.

False negatives are not possible with Bloom Filters, but false positives can happen. This means that if the Bloom filter says that an element is not inside the set, the element definitely is not in the set but if the Bloom filter says that the element might be in the set, one still has to go to the base data and look it up there to get a definitive answer.

To achieve this, Bloom Filters use hashing. When building a bloom filter, a new key is hashed by one or multiple hash functions. For each hash, a bit in a bitset is set. Which bit is set is determined by taking the hash value modulo the bitset length.

In order to test set inclusion the test-key is hashed by the same hash functions that were used to build the filter and the corresponding bitset-entries tested. If one of the

positions is not set, *not-included* will be returned.

Counting bloom filters keep track of how many times some hash value mapped to a position. Therefore, they use an array of counters instead of a bitmap. This increases memory usage but also enables deletes and set-similarity tests [AT06] [GL13].

2.7. Set similarity

Chapter 4 will describe how to use set similarity estimation to check whether a compressed representation of a sample is sufficiently close to its base-sample or has to be recalculated.

The problem of set similarity describes the task of determining how similar two sets A and B are. This can be done using multiple different metrics for their similarity. Simple ones include set-difference $|A/B|$ or $|B/A|$ or the total set difference $|A\Delta B| = |(A \setminus B) \cup (B \setminus A)|$. Another metric for set similarity that is independent of the set size is the Jaccard index or Jaccard similarity coefficient. It is defined simply as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

.

There is also the complimentary Jaccard distance which is defined as $d_J(A,B) = 1 - J(A,B)$ and can also be calculated using the total set difference as

$$d_J(A,B) = \frac{|A\Delta B|}{|A \cup B|}$$

2.8. OddSketches

The OddSketch [MPP14] is a very efficient probabilistic data structure for estimating the total set difference between two sets if those sets are sufficiently similar. For very dissimilar sets, the definition of which depends on the sketch size, the sketch can only say that they are too dissimilar to be able to estimate it.

The sketch has a bitset like bloom filters but only uses a single hash function and uses XOR instead of OR for the bit position. Hence, the corresponding bit for a value will not be set but will instead be flipped. To estimate the total set difference between two sets A and B, the formula

$$|S_A \hat{\Delta} S_B| = -\frac{n}{2} ln\left(1 - 2\frac{|odd(S_A)\Delta odd(S_B)|}{n}\right)$$

derived in the original paper can be used. $odd(S)$ here means the number of set bits in the bitset. This operation can be efficiently implemented using the popcount instruction present on modern CPUs.

The original paper describes doing this for the permutation buckets of a MinHash sketch. However, when the set size is sufficiently close to the number of bits used for the bitset, say $|A| \leq 4n$, the hashes of the original set can be used directly with reasonably good accuracy while still giving a much faster and much more space-efficient approximation of set difference than storing and comparing hashes.

Once the total set difference estimate is available, the union size can be calculated given the set sizes $|A|$ and $|B|$ using

$$|A \cup B| = \frac{|A| + |B| + |A \Delta B|}{2},$$

the intersection size using

$$|A \cap B| = \frac{|A| + |B| - |A \Delta B|}{2},$$

the Jaccard similarity therefore using

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A| + |B| - |A \Delta B|}{|A| + |B| + |A \Delta B|},$$

and the Jaccard distance using

$$\begin{aligned} d_J(A, B) = 1 - J(A, B) &= 1 - \frac{|A \cap B|}{|A \cup B|} \\ &= \frac{|A \cup B|}{|A \cup B|} - \frac{|A \cap B|}{|A \cup B|} \\ &= \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \\ &= \frac{|A \Delta B|}{|A \cup B|}. \end{aligned}$$

The Paper describes a method for estimating Jaccard similarity given the number of permutations in the MinHash sketch if the construction from a MinHash sketch is used.

When using the hashes of the set directly for relatively small sets as described above, OddSketches can additionally handle deletes by just flipping the bit again. Updates can be implemented as a combination of delete and insert. If the base set is too large to use this method directly, it can still be used. This would be accomplished by only updating

the sketch when, for example, the first three bits of the hash are all zero. Consequently, only every eighth element is included which constitutes a form of sampling in itself. Nonetheless, this approach loses the ability to detect very small differences, a limitation likely shared by most other fixed-size sketches for larger base-set sizes. Multi-resolution Odd Sketches [XWZ21] provide a more advanced and more adaptive version of this technique with multiple layers, each with a different sampling probability, whose complexity will not be required for the purposes of this thesis. The idea is similar to that of probabilistic counters for Counting HyperLogLog Sketches described in [FN19].

2.9. Umbra

Umbra [NF20] is a relational research database system developed at TUM which will be used for the evaluation of the implementations described later in the thesis. It aims at achieving the highest possible performance in all scenarios, whether the workload is in-memory or out-of-memory, and whether it is more Online transaction processing (OLTP) or Online analytical processing (OLAP) focused. It achieves this through a combination of modern techniques. A system with goals like these must be written in a low-level systems programming language. In the case of Umbra, C++ is used.

One of the main techniques that makes it achieve its good performance is just-in-time query compilation. What this means is that it will compile a query plan into a single executable binary program that then executes the query. This leads to performance comparable to an optimized hand-written C-program. Umbra uses a more advanced version of the data-centric query compilation model that was pioneered by Umbra's predecessor system HyPer [Neu11]. The technique includes multiple stages of compilation [KLN21] to reduce the impact of compilation times for short-running queries while getting the benefit of aggressively optimized code for longer-running queries.

Being able to parallelize all operations is also mandatory to achieve maximum performance on modern hardware with tens to hundreds of CPU cores. Therefore, Umbra uses a morsel-driven approach also originally developed for HyPer [Lei+14]. Morsel-driven parallelism means that data is processed in chunks which are usually on the order of thousands to tens of thousands of tuples in size. This was later further developed for Umbra to use variable-sized morsels that aim to achieve a predetermined processing time of 2ms [WKN21].

Umbra has an existing implementation of an online sampling algorithm that will be further described in chapter 3 which replaced the original algorithm described in [BRN20]. It is a simpler and more elegant approach that does not have the fixed thread

count limitation and does not need to maintain a concurrent data structure beyond some atomic variables. But has the drawback that it currently does not support deletes (or updates) before this thesis. Besides sampling Umbra also maintains HyperLogLog [Fla+07] [HNH13] and Fast-AGMS [CG05] sketches for each column.

3. Sampling in Umbra

In this chapter, we will describe the sampling method utilized by Umbra. This method is an improvement of the sampling strategy as described by Birler et al. [BRN20]. The sampling strategy remains efficient while eliminating the requirement for an upper bound on the number of threads. It achieves this by incorporating additional ideas from Tirthapura and Woodruff [TW11] and Li [Li94].

3.1. Umbra Online Statistics infrastructure

Before looking at the specific algorithms used for online reservoir sampling in Umbra, it makes sense to talk about the general online statistics infrastructure present in Umbra previous to this work.

Online Statistics is actually in itself a small Hybrid transactional/analytical processing (HTAP) workload. Hence, there are two fundamentally different access paths to it. The first access path, namely point-access updates to the statistics as new tuples are inserted, happens either during transactional processing or bulk-loading. The latter still leads to point accesses because only single tuples in the bulk-loaded data will be inserted into the sample at random positions. The other access path is used during cardinality estimation for almost every SELECT, DELETE, or UPDATE statement and scans all elements in the sample. To optimize for both cases, the cardinality estimation is allowed to be done on slightly out-of-date statistics. Therefore, it uses compressed statistics which are static and can be recomputed from the update-friendly online statistics without the large costs of base table accesses. This is because the online statistics contain all data necessary to compute compressed statistics. Online statistics use a row-wise storage format for sampled tuples so that replacement is fast and localized. Compressed statistics, on the other hand, use a columnar format and offer several pre-computed statistics from the sample and a compressed version of the sample.

One issue that arises from this is of course to choose when to recompute the compressed statistics. Currently, this is done through a counter of insert operations for which a 10% increase compared to the point at which the compressed statistics were created leads to recomputation. In addition to that, recomputation has a hard lower rate-limit per table at once every three minutes.

3.2. Description of the first Umbra reservoir sampling algorithm

In the first online sampling algorithm that has been published [BRN20], an approach was used that relied on a fixed size lock-free linked list. In this list, so-called *skips* were stored. A skip is the number of inserted tuples that need to be skipped before another one is included into the sample. The solution was eventually consistent in that if all n worker threads were inserting into the same relation at the same time, they would be working on the n next skips. But a single-threaded solution would insert them one skip after the other. It was shown that including this sampling logic was scalable and would not significantly slow down the database system.

For deletes, the proposed solution was to increment one of the skips for a tuple that was not in the sample and insert a new skip with skip-length 0 if the tuple was in the sample. For the problem of finding out if a tuple is in the sample and at which position, a full scan of the sample was proposed with the optimization of storing the information on whether the tuple was initially included in the sample as a flag with the tuple. This is of course suboptimal because it is intrusive and needs to be stored with each tuple. Just using scans is also not a good solution because it will significantly slow down deletions, especially in cases where many tuples are deleted at once and in parallel.

3.3. Problems with previous sampling algorithm

The previously described linked list (list of skips) used a fixed-size array of possible list elements which allows it to work on list indexes instead of raw pointers. This also means that, since the list is fixed-size, any index will always be valid. Therefore, no memory reclamation scheme like epoch-based reclamation [Fra04] is required and the data structure cannot grow beyond the initial size.

3.3.1. Problem 1: Bounded number of threads

Because every worker thread could own at most one list element, the size of the list was set to the number of worker threads. This, however, became a problem when Umbra started executing short-running queries on the same threads that receive the SQL statements in the PostgreSQL frontend. There is a possibly unbounded number of those threads.

3.3.2. Problem 2: Deletion

The publication includes a short description of how to handle deletes. Deletes of tuples inside the sample are handled by adding a new element of skip-length 0 to the list

of skips. Deletes that were not in the sample are handled by incrementing one of the skips in the list of skips. This becomes a problem when a large percentage of the data is deleted which will lead to the list possibly overflowing. This situation has no defined handling strategy so some special case logic for it would likely have been required in a real-world implementation.

3.4. New solution: Single node probabilistic adaptation of Tirthapura and Woodruff using Li's Algorithm L

The new method uses a single node adaptation of Tirthapura and Woodruff's [TW11] distributed reservoir sampling algorithm with ideas from Li's Algorithm L [Li94]. The node-local state (u_i) from Tirthapura and Woodruff's algorithm is stored in a thread-local state now. Instead of generating a random number for every newly inserted tuple, a skip is randomly sampled according to the probability of n inserted tuples being skipped with u_i having the value it has like in Algorithm L. This way, processing a single tuple usually becomes just decrementing a single number. That number is then a skip count just like the skips in the original online sampling algorithm from Umbra [BRN20].

Other than in the original algorithm though, this number does not need to be stored anywhere when a local state is destroyed at the end of an insert operation, since it can just be sampled again. To still get the correction of the local u_is, the tuple that is supposed to be taken into the sample from a local perspective, must still get a random number less than u_i assigned to it such that the correction function can still be performed.

The master-node u from Tirthapura and Woodruff is not maintained in an exact way but is instead also updated by sampling a new value as described in Section 2.1.3. For this reason, no synchronized data structure to maintain the nth smallest random number is required.

3.5. Implementation

A thread-local state struct contains the u_i and the amount of tuples that still need to be skipped. After the skip counter is at zero, the next tuple will be included in the sample and a new largest value will be determined by applying a decrease factor of

$$\exp\left(\frac{\log(\text{random}())}{n}\right)$$

with random() being a random value between zero and one and n being the sample size as described by [Li94]. After this, the largest value will be copied into w_{local} of the current thread. Then a new skip count will be determined using the formula

$$\left\lfloor \frac{\log(\text{random}())}{\log(1 - w_{local})} \right\rfloor$$

.

3.6. Missing parts

The Umbra implementation of online statistics is only correct as long as there are only inserts. As soon as there are update and delete operations, the statistics become incorrect. If inserting transactions are rolled back, the sample will not be updated either. Statistics might even become a lot more incorrect than the statistics of systems that just gather static statistics by scanning the data in the background. A quick fix to this could be doing the same by recomputing the sample in the background and then only applying inserts to it again. But ideally, deletes should be handled without requiring recomputation of the sample.

As an example showing the need for also applying updates in the base data to the sample, the motivating example from the introduction (Section 1.2) can be used. Each order has a status field that is initialized with *NEW* for new orders. Most orders in this database will be in status *DONE* and only a very small proportion will be *NEW*. Since updates are not handled, however, the statistics say that all the tuples have status *NEW*. This can lead to wildly inaccurate cardinality estimations for queries having selections on the status field, which will lead to bad query plans being produced by the optimizer.

For deletes, one could imagine a database where the entries in a table are only kept for a week and then deleted. This means that after a year, the sample will claim that the cardinality of the table is about 50 times larger than it actually is. If the table has something like a create-timestamp attribute, the sample will also say that only around $\frac{1}{356}$ of the table contains the previous day as the timestamp even though it should be around $\frac{1}{7}$, assuming the inserts are uniformly distributed over time.

4. Implementation

Random-pairing [GLH06] is a good strategy for handling deletes in samples because it allows for keeping a correct sample without expensive base data access on top of any reservoir sampling algorithm. However, the original work was published in 2006 when multi-core systems were not as ubiquitous as they are today. Therefore, there is also no mention of a way to effectively implement this algorithm on modern hardware. There are several challenges when it comes to doing this. The first one is that just using global atomic counters will cause a large hit to performance since many threads will be fighting over the same cache lines in a parallelized database system. Another challenge is quickly finding out whether a tuple is included in the sample and if so, quickly finding its position within the sample. To test performance in an existing system, Umbra was used for a fast implementation of random pairing. The problem of deciding when to recalculate the compressed sample also has to be solved.

4.1. Checking inclusion of a tuple in the sample

The first challenge is finding out whether a tuple is in the sample or not. The case where a tuple is not in the sample will be the far more likely case for tables with a cardinality significantly larger than the sample itself. Hence, especially this case must be fast for the sample deletion logic not to degrade the overall performance of deletes in the system. Different possibilities were considered for this. Among them linearly searching for each tuple in the sample, ordering the sample in a specific way, for example through Robin Hood hashing, or maintaining an external index of positions.

For deletes, finding the tuple in the sample is only a part of the process. For updates, on the other hand, it is the entire process. In-place updates can just be applied to the tuple in the sample after its position has been found. Out-of-place updates will be handled like a combination of delete and insert by the system anyway.

To uniquely identify tuples in the sample it quickly became clear that the TID would have to be included in the sample. The TID is an already existing 64-bit value that uniquely identifies a tuple within a relation. The TID is generated and returned during the process of inserting the tuples into the specific storage backend and is also known during deletion or updates for all the tuples that are deleted or updated. For other

systems, any unique identifier for a tuple could be used. A unique identifier can usually be derived, for example, through the storage location of the tuple.

4.1.1. Linear search

Linear search within the sample for each deleted or updated tuple is the simplest strategy. Its main benefit is that it is simple and does not require any auxiliary data structures and that it does not slow down the insert process at all.

The problem with this is, however, that it slows down deletes and updates significantly. Since the entire sample will need to be scanned even for tuples that are not in the sample, which will usually be the vast majority, the sampling logic dominates over all other logic that is required to delete a tuple.

4.1.2. Deterministic sample positions

A second approach would be to have deterministic positions for each tuple. This would mean that similar to the way elements are positioned in an open addressing hash table, the elements of the samples would be ordered according to some deterministic criterion, like the TID. The existing tuple could just be replaced at the determined location.

This works well only as long as there are just inserts and updates. As soon as tuples are also deleted from the sample, they create empty spots. Those spots should be filled first since the sample would otherwise take a long time to become full again after deletes. For this purpose, several conflict-handling strategies could be borrowed from hash tables and used for the sample. This results in essentially turning the sample storage into a TID to sample-tuple hash table. Linear probing or similar solutions are not a good fit for this since a miss will be equivalent to a sequential scan. Therefore, some more complex strategies would have to be used. One fitting strategy would be Robin Hood hashing [CLM85]. There are some ways to parallelize this (e.g. [KPM18]) however they all have high complexity and sometimes have to lock a lot of elements at once and also move a lot of elements in memory. Chaining while directly pointing into the sample store is not possible. This would require at least another list containing all the chain head pointers adding additional complexity to the sample.

4.1.3. Auxiliary data structure (Hash Table)

Another approach is maintaining a special data structure on the side that maps the TID to the sample position of the corresponding tuple. For this, all that is required is an implementation of a concurrent hash table and the logic to update this hash table on sample insertions and deletions.

There are many possibilities for implementing a concurrent hash table. One of the simplest somewhat scalable ones is a chaining hash table with bucket-level locking. This simply means that every chain inside the hash table is protected by a mutex.

Compared to deterministic sample positions, this creates another layer of indirection. This is not a problem since the indirection only has to be followed in the case in which the sample contains the deleted or updated tuple, which is assumed to be rare. The benefit of this approach is the simplicity of the inclusion test in the sampling logic and loose coupling of the inclusion test logic compared to the interleaved solution with deterministic positions described before. Performance optimizations to the inclusion test can be made in an isolated component and also tested and debugged in isolation.

For this reason, the approach combines simplicity, performance, and scalability and will therefore be used for all further implementations described. An optimized hash table design will be described in the following section.

Hash table design

To design a good hash table for a given use-case it is important to first make reasonable assumptions about the access pattern. In this case, we can reasonably assume that the vast majority of accesses will be reads and the majority of those will be misses if we assume the base relation to be orders of magnitude larger than the sample size.

A simple global lock for the hash table will not suffice for something that might be called on every delete since deletes are quite fast in Umbra. Hence, a more fine-grained latching scheme must be used. The simplest implementation of this uses a chaining hash table using bucket-level latches and targets a somewhat low fill factor of around 50% so that the chains that need to be walked are not too large.

The case that must be optimized, as described before, is the read-miss case. Having to take a latch for every lookup turns every read into a write which is quite costly. Multiple opportunities for optimization can be applied here without getting into a complex lock-free implementation requiring a memory reclamation scheme.

The first simple optimization that can be made is making the head-pointer of every bucket atomic. This allows for doing an atomic read of the head-pointer which, given a fill factor of 50%, will save around half of the latching for lookup operations already. The other half of the lookups will still need to take the latch and most likely have one cache miss for following the first pointer.

The second optimization, building on top of the first one, is the inclusion of a small Bloom filter in the most significant bits of the head-pointer. Modern CPUs only use the least significant 48 bits of the 64-bit address space when using four-level paging or 57 bits for newer systems using five-level paging, mostly servers. The technique of including additional information in the unused bits of the pointer is called pointer-

tagging. We will be using 8 bits to be future-proof for five-level-paging and since we also do not expect long chains because of the low fill factor anyway. This technique has been described for use in hash tables for hash-joins, which are also prone to having many misses [Lei+14].

The small bloom filter uses a single hash function. So, for each key (TID in this case) that is stored in a bucket, the bit

$$\mathrm{b}(k) = \mathrm{hash}(k) \mod 8$$

will be set. This way a lookup can first atomically read the tagged head pointer and check whether the bit representing the lookup key is set. If this is not the case, we can be sure that the hash table does not contain the lookup key. Only if the bitset contains this key, the latch has to be taken for walking the chain. The drawback of this approach is that for erase operations, the entire chain has to be followed, even if the element to be deleted was already found, and a new updated bloom filter has to be created. This is not significant though since we assume that insert and erase operations will only be a very small fraction of all operations and chains will be short.

4.2. A global-counter based implementation

One first reference implementation is random-pairing based on two global counters that are always updated for every insert (if they are not both 0) and every delete. This can be done either using atomics or using a global mutex. A global mutex is the simplest implementation. It will, however, most likely lead to contention in some cases. Atomics are only slightly better because they also lead to contention. They just allow optimistic operations to go on in parallel.

The two counters are held as members of the OnlineStatistics class which will also have the implementation of random pairing. On each delete of a TID that is an element of the sample, the deletedFromSample counter will be increased. In case the deleted tuple was not an element of the sample, the deletedWithoutSample counter will be increased.

For inserts, they will apply the random-pairing logic if the sum of both counters is greater than zero and decrease the corresponding counter. When using atomics this needs to be done with care since the counter could have become zero which means atomic compare and swap has to be used.

This solution will work fine for a small number of threads. Accessing atomics in a highly parallel context, however, will lead to contention because of many threads accessing the same cache lines.

4.3. Introducing Local states

Thread-local states can be used to reduce the contention on the global counters. The thread-local state will be initialized before a pipeline starts being executed by the execution engine and finalized when the pipeline is done processing.

4.3.1. Parallelizing deletes

Each deletion processing pipeline will have a local state for deletes that includes local counters for deleted values from and not from the sample. These counter values will be merged into the global counter after a thread has finished processing the deletes of a pipeline. This way, fast local adds can be used most of the time. The fact that there could be parallel inserts that would then still be using the regular sampling logic is not problematic since one could imagine all those inserts occurring before all the deletes in the timeline which would produce the same result.

Figure 4.1 shows an example execution with two threads starting some deletes at different times. The changes in the thread-local states of both threads are shown on the side beside the deletes. After they are finished with their deletes, the thread-local state is merged into the global state that is assumed to be initialized with both counters set to zero before the start of both threads. Then the thread-local state of thread 0 is merged into the global state after which the local state of thread 1 is merged.

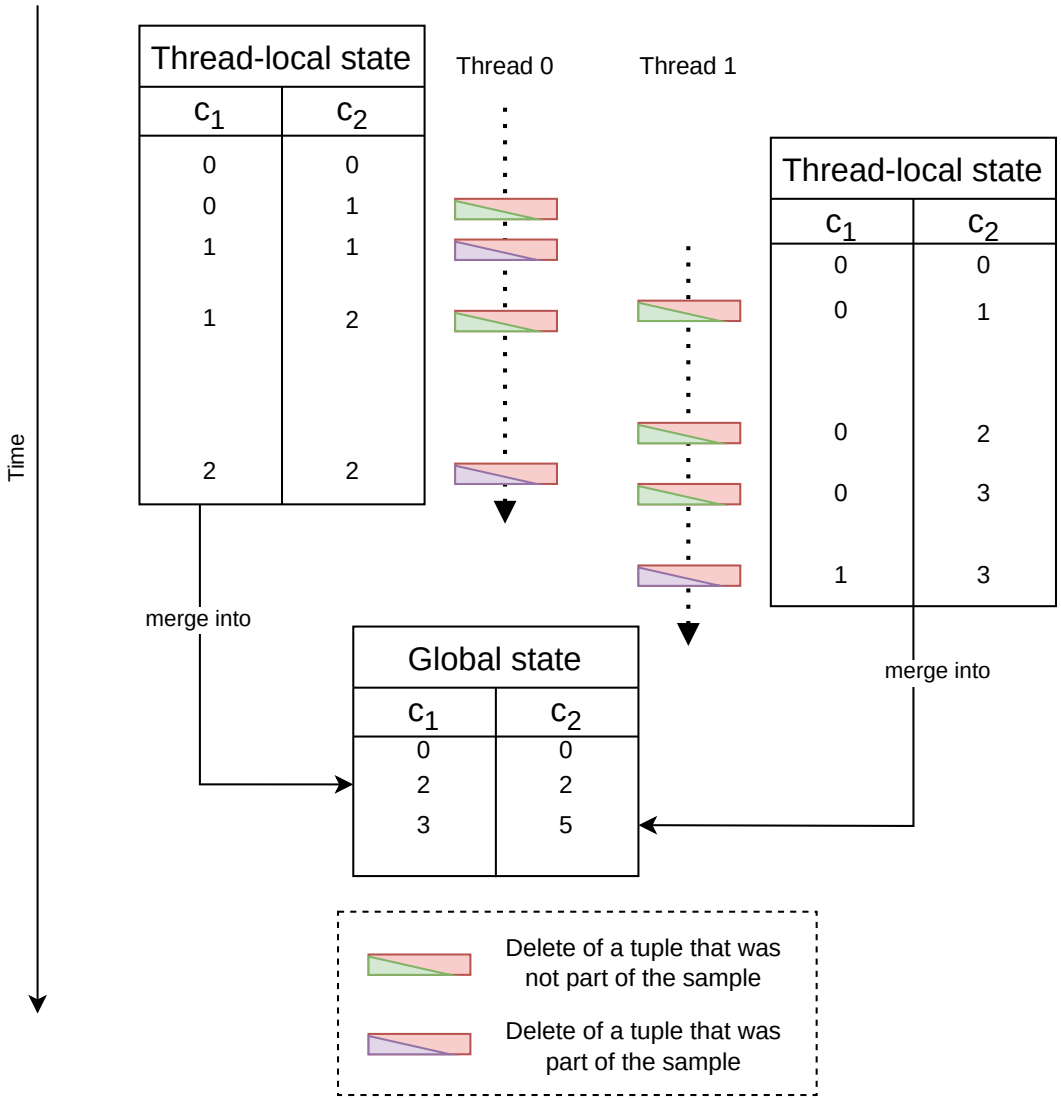


Figure 4.1.: Implementation of random-pairing deletes in a multithreaded context. Assuming global state before both threads start executing is initialized with 0.
c_1 counts the deletes from the sample, c_2 counts the deletes that were not in the sample at the time of the delete.

The free positions are managed in a lock-free queue from the boost library [Boo] but could generally be managed in any concurrent data structure that allows for any kind of push and pop operations. The order in which elements are output by pop is

irrelevant. Most likely even a queue secured by a single lock would be acceptable since it is only accessed when the sample actually changes.

4.3.2. Parallelizing random-pairing insert logic

Parallelizing the inserts is not trivial. One could again use the global counters for every insert but if there has been a large amount of deletes, this would mean a lot of contention that would slow down inserts. Therefore, each inserting thread will split off some fraction of both counters and process those locally before checking the counters again.

Splitting off the same fixed fraction of both counters is stochastically incorrect because theoretically, in a single-threaded case, all inserts into the sample could happen within the first 1% of the random-pairing inserts. Hence, only one counter can be split in this way and the proportion of the other counter that is used must be sampled from a random distribution. The counter that will be split fractionally is the larger of the two counters. This will usually be the counter that counts deleted tuples that were not in the sample except for the case in which pre-loading is still in progress while deletes happen. The suitable distribution would be a binomial distribution because the random experiment can be modeled as having an ordered list of non-sample-deletes and then for each sample delete deciding whether it was before or after the threshold. Alternatively, a skip length could be sampled. This optimization was not done for now since even the performance impact of the more naive random pairing is minimal.

Figure 4.2 shows an example execution with two threads inserting after previous deletions. The changes in the thread-local states of both threads are shown on the side beside the inserts. The global state starts with one deleted tuple from the sample and five deleted tuples that were not a part of the sample. The larger of the two counters (c_2 in this case) is split up equally among the two threads (three and two). The splitting of the second counter, on the other hand, is done randomly as described before, which leads to the one delete from the sample going to thread 1 in this example. Thread 1 inserts four tuples which means that its local counters are both zero after three of the inserts. Thread 0 will then look at the global counters and, after seeing they are also both 0, it will continue with the normal reservoir sampling logic. Meanwhile, thread 1 only inserts two tuples. The first one is randomly decided to replace the deleted tuple from the sample and the second is therefore ignored for the sample. After thread 1 is done with its inserts, there is still a local counter value greater than zero so they are merged back into the global state. If a third thread were to start inserting while the other two threads are still working, it would just do its normal reservoir sampling logic. It would then look at the counters again after it placed a value into the sample in this way if it inserts enough values for this to happen.

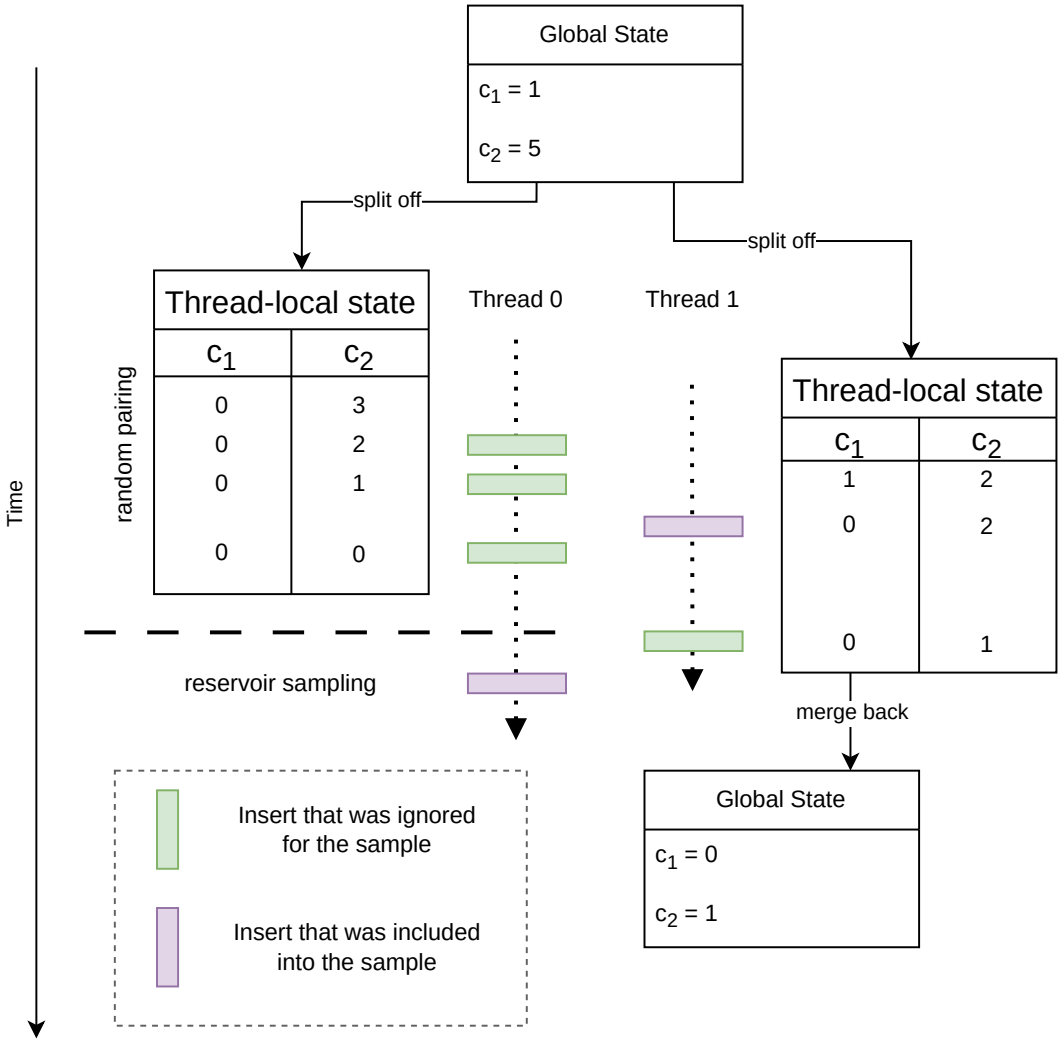


Figure 4.2.: Implementation of random-pairing inserts in a multithreaded context. c_1 counts the deletes from the sample, c_2 counts the deletes that were not in the sample at the time of the delete.

4.4. Reducing contention for small transactions: Introducing shards

If every transaction only inserts or deletes a small number of tuples the benefit of the local states is very limited. In the extreme case, where every transaction only touches one tuple, the benefit might even be negative since splitting and merging are two operations on the global counters that would have just been one operation if it had been applied to the global state directly.

4.4.1. Solution using shards

To counter this problem, the global state can be split into a number of shards. The number of shards should be chosen in proportion to the expected degree of parallelism or in proportion to the hardware concurrency of the machine that is running the workload. The shards, while still being global and therefore synchronized (using atomics/mutexes) state, are then assigned threads according to some thread identifier (thread-id).

4.4.2. Assigning Thread-ID's

There are multiple possibilities for getting such a thread-specific number. One is to use the thread-id C++ provides using std::this_thread::get_id. However, these IDs are not ideal for this purpose since multiple of them being assigned the same shard is very likely as it can be assumed for this purpose that they are just uniform-randomly distributed. This would then again lead to multiple threads accessing the same cache lines which should be avoided as much as possible. A much better choice is worker-thread-IDs if some kind of scheduling system already provides them. Umbra does have such a scheduling system, which also provides contiguous worker thread IDs starting from one. Assigning those to shards would be perfect if no other threads were accessing the counters since there would be no contention and no shared cache lines. There are also other threads active, however. Umbra executes small transactions on the network connection threads directly instead of scheduling them to the workers. Thus, there is still a need to assign IDs to those threads. One approach that should work sufficiently well here is a global counter that is used to assign increasing IDs to a thread-local variable for each thread. The global counter would only be touched once in a thread's lifetime and should therefore not be a major source of contention unless a new thread is created for every transaction which will have a large overhead anyway. The shard index to use is then determined by taking

$$shardIndex(shardingId) = shardingId \mod shardCount.$$

The sharding approach also removes the need for the splitting logic in inserts since deletes already produce counters split among the shards. It also does not introduce a limitation for the maximum number of threads or anything similar to it. Figures 4.3 and 4.3 visualize the delete and insert logic now including shards.

4.4.3. Balancing shards using shard-stealing

Sometimes shards could become stale or unbalanced. The user could, for example, set a different desired degree of parallelism. Worker threads could also die or starve in some cases, meaning some shards might also not have a corresponding active thread to use its random-pairing counters for inserts. Another scenario is some threads generally getting more deletes while others get more inserts.

Therefore, it is important to balance out different shards at least eventually. To achieve this, a small percentage like 5% of all insert operations does not go to its assigned shard but does instead go to a random shard to *steal* its counters. This does not significantly increase contention on any single shard. If the insert operation then finishes without using up the random-pairing counters, it will merge them back into its assigned shard and not the shard they were read from. This covers the cases where a shard does not have an active assigned thread anymore and the case where one thread gets significantly more inserts and another gets significantly more deletes.

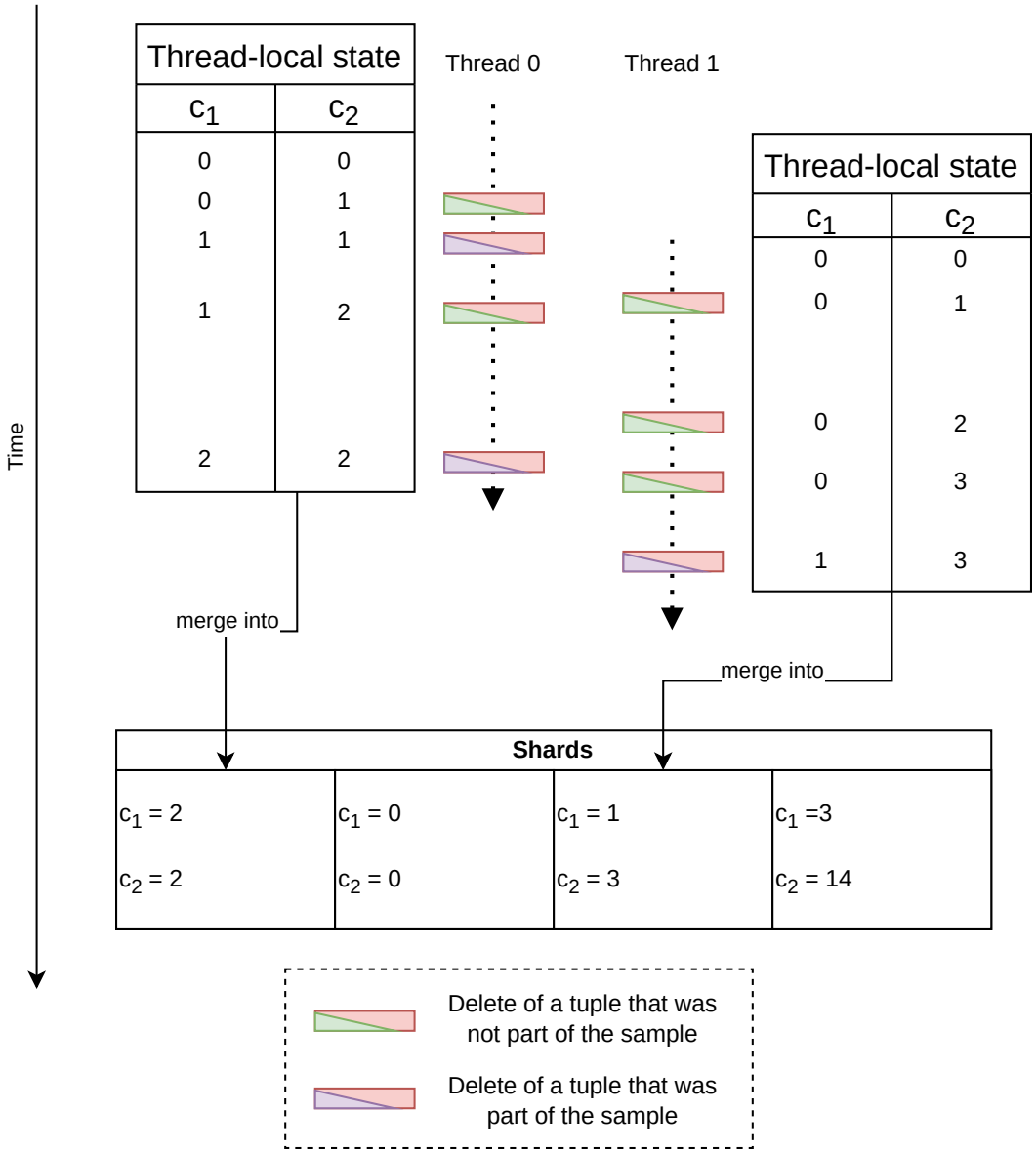


Figure 4.3.: Implementation of random-pairing deletes in a multithreaded context using a sharded implementation.
Assuming counters in both threads shards before both threads start executing are initialized are 0.
c_1 counts the deletes from the sample, c_2 counts the deletes that were not in the sample at the time of the delete.

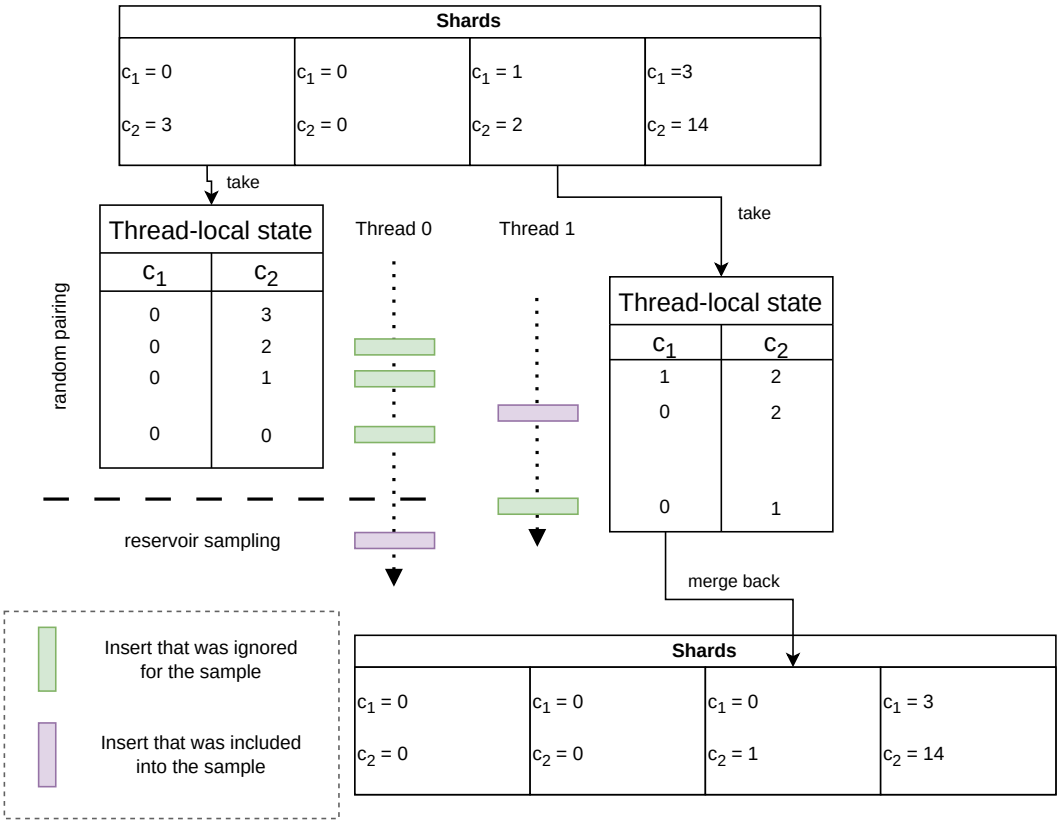


Figure 4.4.: Implementation of random-pairing inserts in a multithreaded context using a sharded implementation.
c_1 counts the deletes from the sample, c_2 counts the deletes that were not in the sample at the time of the delete.

4.4.4. Replacing preload with random pairing

Reservoir sampling techniques usually assume an already filled sample. Therefore, previous approaches implemented in Umbra all required special *preload* logic. Preload describes the phase during which the relation size is less than the maximum sample size. This means that during this phase the sample is equivalent to the entire relation. Preload was previously done using an additional counter that was checked before using the actual sampling logic.

When also considering deletes, preload becomes a harder problem, however, since the relation size can drop below the maximum sample size an arbitrary number of times. For this reason, there is no clear preload phase that will be exited only once like there is when only considering inserts.

The special handling of preload can be avoided by initializing the random-pairing counters in a way that is equivalent to having a full sample and deleting all values out of it. When using the sharding approach, the value should be distributed among all the shards. The free position list must then also be initialized with all positions.

This does conceptually work, however, the online reservoir sampling algorithm in Umbra is in a separate component that is used not just in the online statistics but also for approximate query processing. The original preload mechanism is not really problematic anyway since it is very predictable for branch prediction and did not have any noticeable impact on any benchmarks. Therefore, the old mechanism was kept in place and the random-pairing preload was not included in the final solution for Umbra. If another system were to implement reservoir sampling logic just for the statistics use case, it should not implement special preload logic and should instead use the method described above.

4.5. Handling Transactions

Having a transaction-specific sample causes significant implementation complexity while most likely leading to some performance degradation with very limited benefit in statistics quality. The only case where it would actually be beneficial to have a sample for a single transaction would be when bulk-loading data while other transactions are executing expensive and complex OLAP queries. This case could be limited without having to make the sample truly transactional by keeping to use the current compressed statistics instance while a significant amount of sample tuples has been inserted by uncommitted transactions or more simply as long as any bulk loading transaction is in progress. Umbra already keeps track of bulk operations [FKN22] for its Multi-Version Concurrency Control (MVCC) implementation.

4.5.1. Recovery

At normal shutdown, which happens only after all queries have finished processing, it is easy to just save the random-pairing counters together with the rest of the sampling state. At shutdown, all random-pairing shards or global state (depending on the chosen implementation) will be fully present in the class members of the online statistics and can just be serialized and written to disk.

Crash Recovery is not as easy to implement, however, because it cannot be assumed that no thread is accessing the state at any point. This would be required to get a consistent snapshot of the sample. It could theoretically be handled by also writing all sample changes to the write-ahead log and having the checkpointing process also apply all changes to some on-disk representation of the sample from time to time. Since statistics should be considered as a whole in this issue, however, this is not an option because this solution does not work for sketches. Also, the random-pairing counters will sometimes be moved to thread-local states for processing and therefore not be present or accessible through any members of the online statistics class. This is also not desirable since it would require making the thread-local state atomic again, in which case the shards of the sharded implementation could just be used directly which will lead to performance degradation in the case of bulk-deletes for example.

Approximate solutions

Since crashes are assumed to be rare, resampling and recalculating other statistics once at startup after a crash is a reasonable solution. An imperfect but workable solution should also be to get a potentially inconsistent snapshot of the statistics periodically through the checkpointing process and save this to disk, in which case normal recovery could be applied. The snapshot would just read the atomic values one by one and save them to a copy. This could mean that a tuple is not applied to HyperLogLog sketches in the snapshot but is already in the sample, for example. Then, on recovery, all the log entries appearing after the one describing the checkpointing of the online statistics could just be applied to the statistics. In an implementation that uses thread-local state, it would probably be necessary then to randomly resample enough tuples for the sample such that it is full again since there is no good way to recover the random-pairing counters in this case. As it can reasonably assumed, that usually only a small portion of the sample will be deleted at any point in time, the time necessary for doing this should not be high in the average case.

Exact solution

An exact solution would require consistent snapshots of the statistics through the checkpointer. This could be achieved by having a checkpointing read-write latch for every online statistics instance on which each query takes a read-latch before starting with the processing of a query. A thread would then only be allowed to hold thread-local state if it also holds a read-latch for the respective online statistics object. The checkpointer would then take a write-latch here. A fair latch would need to be used to ensure the checkpointer does not starve for tables with a constant stream of transactions operating on them. Then, however, transaction processing on that table would be stopped for a short amount of time. One way to mitigate this is to have an operation buffer for this case. It buffers all operations that need to be applied to the statistics when the lock is released again. The checkpointer would then apply all the buffered operations after it is done taking the snapshot of the statistics.

4.6. Binary Format

The compressed statistics C++ object contains all the information necessary to reconstruct the online statistics it was created from. The sample is serialized into a custom binary format. This format previously assumed that the sample is either full or in preload, in which case the positions of the tuples within the sample can be reconstructed through the bijective position mapping used for preload.

This format needs to be augmented to be able to handle deletes because deletes can cause a sample to be neither full nor in preload. To achieve this, first, a bitmap of max-sample-size bits (rounded up to the next 8 bytes) representing a present map for the sample positions is stored after the 32-bit header. After this, the TIDs of the tuples are stored in the same order as the tuples will be stored. Afterward, the tuples are stored as before. This serialization process does not take any kind of global lock on the online statistics data structures. Therefore, when copying the sample, it is possible that the space initially allocated for the sample is not sufficient to actually fit the entire sample. In this case, the last sample elements that do not fit anymore are simply ignored. In the case that there was too much space allocated, the validSampleElements counter in the header will make sure that any reader will only read the actual existing number of elements.

4.7. Random replacement

A solution for deletions that seems obvious is sampling a random tuple from the base data to replace a deleted one while keeping the sample full.

This is not as easy as it first seems, though. The first thought one might have is sampling a TID and just reading that. Since TIDs are not dense however, e.g. because of deleted tuples, the sampled TID will very likely not even exist if there have already been many deletes before.

The next idea one can come up with is doing a random walk through the B+Tree, which Umbra uses for storing base-tables when using the *paged* storage-backend. This is mathematically incorrect, though, because at each page the fanout can be between the maximum fanout and one-half of the maximum fanout as described in Section 2.3.2. Therefore, the chance of picking a particular child node can be different by a factor of 2 between two pages. Leaf pages can also contain variable-sized data. Even if we assume that we can uniform-randomly pick a leaf node we still can have the case that one leaf only contains one large tuple whereas some other page contains hundreds of small tuples. In this case, the large tuple has a chance of being picked that is hundreds of times larger than each of the small tuples on the second page.

Through the B+Tree uniform-random sampling algorithm described in Section 2.3.2 a random leaf page can be sampled. Variable-sized keys are not handled by the approach. A small extension using the same acceptance/rejection sampling method can be used for variable-sized keys. We just calculate the maximum number of tuples in a leaf page by assuming all variable-sized data has a size of 0 and then perform acceptance/rejection sampling using this number. So we sample a random number out of the previously calculated maximum tuples per page and restart the random sampling process from the root if the sampled number is larger than the actual number of tuples on the leaf. This should be correct but can become very costly and lead to many restarts if the average size of variable-sized data is much larger than the fixed-size part of the tuples. If the tuples are 20 times larger than the fixed-size part of the tuple, it will on average take 20 walks down the tree until a single leaf-tuple can be taken. This will not be skewed access to B+Tree pages like most real-world workloads but will instead be random access. Unless most of the B+Tree is in memory anyway, this will cause a lot of slow random I/O.

If a table has some index with fixed-size keys, which will often be the case for primary key indexes, this index could be used instead of the relation-storage B+Tree for faster sampling with fewer restarts. If random replacement is done in any form, it might be useful to keep a running average of the average number of retries or average time required to get a random tuple and only do random replacement with a probability inversely proportional to this number.

Since the relation size may drop below the maximum sample size again at some later point in time, some form of random pairing is still required to remain correct. The only other possibility is to reset the reservoir sampling algorithm into preload, scan the base data, and insert every tuple into the sample.

Another problem that needs to be solved is avoiding duplicates in the sample. To achieve this, an approach like the concurrent hash table described earlier can be used. If there is a hit in this hash table, a new tuple needs to be sampled. If there are some number of hits in the hash table in the retries, it must be assumed that the relation size is close to or below the maximum sample cardinality so retries must be canceled at some threshold of maximum retries to not run into an endless loop of resampling when the relation cardinality drops below the maximum sample cardinality.

4.7.1. Hybrid approach of random pairing and random replacement

As described in the previous section, random replacement is not desirable because sampling a random tuple can be relatively slow depending on several factors like storage type, disk/network latency, and restart probability. However, random-pairing alone has the issue of leading to bad sample quality due to the sample cardinality significantly shrinking when a large percentage of the base data is deleted. When 90% of the base relation is deleted, most likely also around 90% of the sample is deleted, meaning the chance of a tuple being in the sample is 10 times lower than if the inserts that were later deleted never happened. A hybrid approach can be used to mitigate this issue for this degenerate case while not taking the potentially high cost of random replacement in the non-degenerate case where inserts and deletes at least balance out. Some target lower-bound for sample cardinality is defined. The sample deletion algorithm always tries to keep the sample cardinality above this threshold through random replacement unless the cardinality of the base relation drops close to it or below it. 50% of the maximum sample size would probably be a reasonable threshold. In any other case, just random pairing is used.

To decrease the impact of random replacement further here, it could be delegated to a background worker that only does random replacement. Through the setting of the maximum number of such background threads or background tasks, the impact that this has on overall system performance could also be bounded or those tasks could be shifted to a time of lower system load.

4.8. Analysis

Almost all the analysis necessary has already been done in the papers describing the distributed reservoir sampling approach [TW11] and random pairing [GLH06]. There

are a few implementation-specific details for the parallel implementation that still need to be looked at.

4.8.1. Insert

For inserts, all the operations are constant-time except for the random-pairing inserts. Those are generally also constant-time in the originally described variant and also the sharded variant which will simply have to look into its assigned shard and either move the counters to the local state if they are larger than 0 or do nothing.

4.8.2. Update

For updates, all that is done in addition to what has to be done to execute the update is a single lookup in a hash table and potentially the replacement of the value in the sample. The hash table lookup is $\mathcal{O}(1)$ and the replacement is $\mathcal{O}(t)$ with t, depending on the implementation, being either the length of the old and new tuple or the length of some representation of the update. This is also unproblematic and unavoidable.

4.8.3. Delete

For deletes, there is only the $\mathcal{O}(1)$ lookup from the hash table and potentially the $\mathcal{O}(1)$ delete from the hash table. The tuple will actually stay in the sample storage and will only be overwritten, with string data freed when a new element is inserted in its position.

4.9. Accurate Sample similarity estimation

An additional problem arises when applying deletes and updates to sampling, which is that a new method for finding out how much the online sample deviates from the compressed sample needs to be developed since the old implementation just used the ratio of total inserts in the online statistics and compressed statistics which means that queries will keep using old compressed statistics if there are only updates and deletes but no inserts.

A simple solution would be to turn the insert counter into an operation counter including update and delete operations. This can lead to unnecessary sample recompression if rows are just changed between two values all the time. As an example for this, one could imagine a parking lot occupation tracker that just uses a single table for parking spots with an *occupied* boolean column. The only time the statistics will actually need to be recalculated is when the set of occupied parking spots changes

significantly. An operation counter will not notice that some of these changes will be reversing other operations. For this reason, a better solution can be found.

One could build an array of all sample-tuple hashes, or some part of all those hashes and put them into an array but this would have significant memory usage of at least 1kB for one byte of each hash and a sample of size 1024. Therefore, a more compact representation is desirable since the comparison would ideally be done for every query being optimized.

4.9.1. OddSketches for estimating differences

A natural solution for a problem like estimating the similarity between two sets is using some kind of small sketch. The sketch used in this case should have a few properties:

1. Be able to estimate total set difference or Jaccard similarity
2. Be cheap to maintain
3. Have low memory footprint compared to an array of 8-bit hashes of sample elements (ideally at most one bit per sample element)
4. Support deletes without recomputation of the sketch
5. Be able to estimate set difference very cheaply so that the cost of this operation is insignificant compared to the cost of compressed-statistics recomputation

No existing sketch directly fulfills all the requirements. MinHash sketches, b-bit MinHash sketches, or Counting Bloom Filters (using counters instead of bits) could be used but MinHash sketches do not support deletes. For Counting Bloom Filters with 16-bit counters, as required to support the full 0-1024 range, this is only at most 64 counters. This will, according to small synthetic benchmarks, not lead to good estimation performance when using the estimation method from [AT06].

One method that achieves good results at just 0.5 or even 0.25 bits per sample element is using OddSketches [MPP14]. To support updates and deletes directly, the modified form described in Section 2.8, which constructs an odd sketch directly from the data, must be used.

OddSketch implementation

The implementation of an OddSketch is simple. Updates to the sketch, both inserts and deletes, are just flipping the bit corresponding to their hash as can be seen in Listing 4.1. The C++ Class is templated so that the size of the sketch and the base data type

can be specified by the user of the class. This also makes it possible to use atomics instead of regular integer types.

```
void update(uint64_t hash) {
  auto pos = hash & hashMask;
  auto data_index = pos >> (baseShift + 3);
  auto bit = pos & bitMask;
  data[data_index] ^= (1ull << bit);
}
```

Listing 4.1: Update (Insert/Delete) operation for OddSketches implemented in C++. inverseBitlength is a precomputed constexpr of 2/#bits

Estimation of the difference between two sets represented by two OddSketches is just a matter of summing up the number of 1-bits in the xor of both sketches. This can simply be implemented like in Listing 4.2. The *r0* value can be less than zero which would lead to the logarithm being undefined. Therefore, an optional is returned by the function, which is empty if this happens. This will usually only happen for very large differences for which we would want to recompress the sample anyway.

```
std::optional<double> estimateDifference(const OddSketch<shift, base>& other) const {
  uint64_t combinedOnes = 0;
  for (size_t i = 0; i < (1ull << actualShift); i++) {
    combinedOnes += std::popcount(data[i] ^ other.data[i]);
  }
  double r0 = 1. - static_cast<double>(combinedOnes) * inverseBitLength;
  if (r0 <= 0) {
    return std::nullopt;
  }
  return -static_cast<double>(bitLength) / 2. * std::log(r0);
}
```

Listing 4.2: Estimation procedure for two OddSketches

Jaccard distance can then later be estimated by first calculating the size of the union of both sets

$$|s_1 \cup s_2| = \frac{|s_1| + |s_2| + |s_1 \Delta s_2|}{2}$$

and then the Jaccard distance using

$$d_J(s1, s2) = \frac{|s_1 \Delta s_2|}{|s_1 \cup s_2|}.$$

The Jaccard distance can then simply be compared against some threshold like 5 or 10%.

The estimation process could be manually vectorized using 4 SSE (128 bit), 2 AVX2 (256 bit), or one AVX512 (512 bit) vector for the x86_64 instruction set. However, where it actually makes sense, both GCC and Clang will auto-vectorize the xor and popcount operation.

```
1  xorpopcnt(std::array<unsigned long, 8ul>&, std::array<unsigned long, 8ul>&):
2  vmovdqu64 zmm0, ZMMWORD PTR [rdi]
3  vpxorq zmm0, zmm0, ZMMWORD PTR [rsi]
4  vpopcntq zmm0, zmm0
5  vextracti64x4 ymm1, zmm0, 0x1
6  vpaddq ymm1, ymm1, ymm0
7  vmovdqa xmm0, xmm1
8  vextracti64x2 xmm1, ymm1, 0x1
9  vpaddq xmm0, xmm0, xmm1
10 vpsrldq xmm1, xmm0, 8
11 vpaddq xmm0, xmm0, xmm1
12 vmovq rax, xmm0
13 vzeroupper
14 ret
```

Listing 4.3: Assembly generated by GCC for counting the 1-bits in the result of xoring two OddSketches with 64-byte length

When setting the AMD Zen4 architecture in GCC through `-march=znver4`, for example, GCC 13.2 using `-O3` will produce very short and well-auto-vectorized code using just very few AVX-512 instructions to achieve the task as can be seen in Listing 4.3. Within this assembly code `vmovdqu64` loads the entire sketch at once into a 512-bit register, `vpxorq` does the bitwise xor and `vpopcntq` does a popcount per 8-byte integer in the vector. The instructions from line 5 onward just do a horizontal add which adds all these popcounts. Setting a modern Intel architecture like `-march=icelake-server` will produce similar code. Since, for example, the `vpopcntq` instruction is only available for CPUs with AVX-512 support, this will not work this efficiently on older CPUs and will only be able to utilize SIMD instructions in a more limited way. A collection of different configurations and the produced assembly for both a current GCC and a current Clang version can be found in the appendix (Chapter A.1), where also the less vectorized code on older CPU architectures can be seen. Since none of the assembly snippets contains any branches, execution will be able to take advantage of superscalar CPU architectures rather well which means that the CPU will execute multiple instructions, in this case multiple XOR or popcount instructions, at once even without SIMD. There will also be

at most one cache miss per sketch since a 64-byte OddSketch fits into a single cache line and will also be aligned to one cache line.

Auto-vectorization will only happen when both sketches are not using atomics because, otherwise, the compiler will assume that the data in the other atomic fields might have changed during the previous loop iteration. Listing 4.4 shows this for GCC. Clang will still auto-vectorize the code but will insert a lengthy sequence of instructions for initializing the 512-bit vector one 64-bit atomic value at a time before executing the AVX-512 instructions. In the implementation for checking similarity between the online statistics sample and the compressed sample, the latter will have a non-atomic sketch because it is read-only whereas the former will have an atomic sketch using `std::atomic<uint64_t>` for its data. Thus auto-vectorization cannot be utilized when simply adding `other.data[i].load(std::memory_order::relaxed)`. The work that needs to be done is still just 8 loads, 8 xors, 8 popcounts, and 8 conditional branches for the loop (lines 5-12). Hence, the performance impact of this for work that is only done once per table per query-optimization run should be minimal.

```
1  xorpopcnt(std::array<unsigned long, 8ul>&, std::array<std::atomic<unsigned long>, 8ul>&):
2    xor edx, edx
3    xor ecx, ecx
4    .L2:
5      mov rax, QWORD PTR [rdi+rdx]
6      mov r8, QWORD PTR [rsi+rdx]
7      add rdx, 8
8      xor rax, r8
9      popcnt rax, rax
10     add rcx, rax
11     cmp rdx, 64
12     jne .L2
13   mov rax, rcx
14   ret
```

Listing 4.4: Assembly generated by GCC for counting the 1-bits in the result of xoring two OddSketches with 64-byte length

As the result is only approximate anyway, it does not matter whether the reads for similarity comparison will be slightly off or not atomic in this sense. Therefore, one could consider casting the atomic sketch to a non-atomic sketch for the calculation. This will technically be undefined behavior in C++ because it can lead to data races, however, in reality, at least on x86, it will lead to acceptable behavior since the sketch fits into and is aligned to one cache line exactly which means the read of the 512-bit vector will most likely behave almost like an atomic read of the entire sketch at once. Even if the sketch was larger than 512 bits and therefore crossed cache line boundaries, the worst

that could happen is a torn read which is completely acceptable for the bitmap of an OddSketch.

4.9.2. Building combined hashes for a sample tuple out of column hashes

The last part missing for a complete solution is how to create the hashes for inserting into the OddSketch. Just using the TIDs of the tuples or a hash of the TIDs is not enough since updates would not be considered for similarity then. Another thought would be to use a hash of a combination of TID and some sort of timestamp using either just some actual 64-bit timestamp like the value of the *RDTSC* register on x86_64 processors or the Log Sequence Number (LSN) of the operation that last modified the tuple. This is however only slightly better than using the operation counter.

A more optimal way to get hashes that actually represent the changes to the tuple content is to use a combined hash of the single-column hashes already created for Sketch updates. A basic form of combining the hashes would be to just use the bitwise-XOR operation on all the single-column hashes. This means, however, that if two columns swap value the combined hash will be identical. To prevent this, the hash of each column h_i can be multiplied with some unique factor a_i before combining them. If a_i is co-prime to 2^{64} (for 64-bit hashes) the result will even be without loss of information. This is because a multiplicative inverse in $\mathbb{Z}_i$ will exist for any number co-prime to i. Finding a generator function for a series of such numbers $f_a(i) = a_i$ is quite easy. The function for the i'th odd number $f_a(i) = 2i + 1$ can simply be used. Any odd number is co-prime to any power of two since no odd number will have two as a prime factor while any power of two will only have two as a prime factor.

The hash of the entire tuple will then simply be stored in the header of each tuple in the sample to allow for fast deletions from the sample and therefore the OddSketch. The major benefit of combining hashes in this way is that updates to a single column can be performed using just the previous combined hash and the single-column hash for the updated column both before and after the update. The single-column hashes can then simply be multiplied with a_i and both applied to the sketch.

5. Evaluation

To assess the performance characteristics of the different solutions proposed, several benchmarks will be used for comparing their implementations.

5.1. Correctness

Before evaluating the performance of the implementations, it is important to check whether the implemented logic leads to correct results. To ensure correctness, the Anderson-Darling-Test [AD52] was used. The sharded implementation was tested since it represents the most advanced variant presented in the previous chapter.

All tests were run with the current hard-coded Umbra setting for sample size, which is 1024. Different configurations of base data size and parallelism were tried in a fashion similar to [BRN20] just with an additional Anderson-Darling-Test step after chunk-wise deleting and reinserting half of the data. The single-threaded test is always correct. The multi-threaded results can turn out differently over multiple runs of the test because the shard-stealing described in the previous chapter leads to shards sometimes being empty and others being overly full. Therefore, the random-pairing counters might not all be 0 after the reinsertion as is the case in the single-threaded test. Over time and with more inserts than deletes this would balance out, however.

5.2. Setup

All performance measurements were made on a machine with 1TB of main memory and 4 Intel® Xeon® X7560 (2.6GHz) Processors with a total of 32 cores (64 threads). Usually, if there is no further information provided, it can be assumed that benchmarks will be fully in memory. For the Umbra-based benchmarks, it can also be assumed that the write-ahead log will be written to an in-memory file system to ensure that writing the log to disk will not be the bottleneck in the benchmark. Umbra-based benchmarks all use the *paged* relation storage backend which is focused more on OLTP and HTAP workloads and uses a per-page PAX layout as described in [NF20].

5.3. Microbenchmark

To test the performance impact of the changes made, a small microbenchmark was created. The benchmark is parameterized by the number of tuples, the number of threads, and transaction size, which is the number of insert or delete operations done before the local state (if applicable) is finalized and a new one is created. The transaction size is supposed to test use cases that are more like OLTP use and others that are more like OLAP use with bulk loading in the form of ETL processes or even loading the entire dataset at once.

The microbenchmark first inserts the specified number of tuples into the online statistics in batches of the specified transaction size and then deletes them again. 10 Million was used as the number of tuples here which translates to around 8000 actual insertions into the sample. Around 5000 of those happen within the first 10% of inserts and 3000 of those happen within the first 1% of inserts. It can therefore be assumed that the performance measured here mainly shows the performance for the decision logic. Since there is nothing to do for the reference implementation (which has no delete logic) and it is also unrealistic that a delete does not require any additional action, a deleted-bit is set in a shared atomic bitmap for a minimal delete workload in addition to the sample delete logic. The TIDs are shuffled before deletion in order to prevent too much contention on the bitmap.

It does not make sense to compare random-pairing- and random-replacement-based implementations at this microscopic level since the performance of random replacement mainly depends on the time needed to fetch a tuple randomly.

For deletes, it can be seen in Figure 5.1 that up until a transaction size of about 10, the local state implementation is almost equally bad as the global state implementation because the thread-local state has very little benefit there. After that, the benefit of the sharded implementation becomes insignificant and the thread-local state optimization dominates, meaning that for transaction sizes of 100 or more, the local state and sharded implementations have almost identical performance characteristics.

For inserts without any prior deletes (Figure 5.2), especially at larger transaction sizes, the updates of sketches dominate performance. This effect can also be seen when profiling Umbra. Hence, Figure 5.3 shows the numbers with sketch updates deactivated. This shows that inserts with random pairing do more work and have more branches but are still in the same order of magnitude as the reference, which is in the hundreds of millions of operations per second just like deletes. At smaller transaction sizes (<=8 operations per transaction) all implementations have significant overhead. However, their performance is all within the same general range.

For inserts that happen after deletes (Figure 5.4), where the non-reference implementations considered here are using random pairing, there are significant differences again.

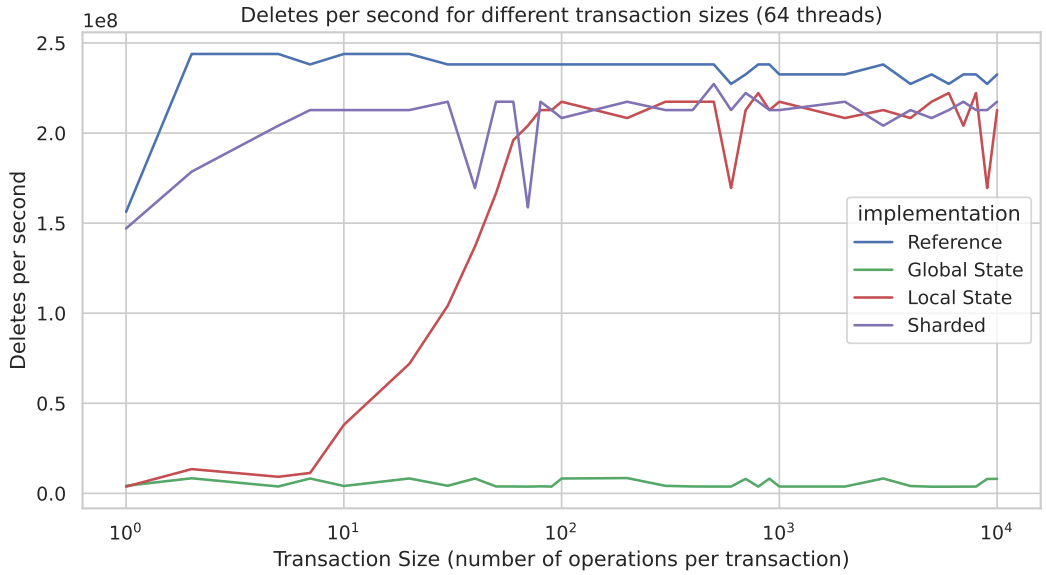


Figure 5.1.: Microbenchmark results: Deletes per second at 64 threads depending on transaction size

At transaction sizes below 100 random-pairing insert operations per transaction, the local-state-based implementation without sharding has significantly worse performance than the sharded implementation which performs only slightly worse than the reference without sample-deletion logic. The local state implementation can only perform 50 thousand inserts for transaction size 1 whereas the sharded implementation can do 2 million inserts in this case. Even the global state implementation does much better up to a transaction size of around 20.

Without sketch updates, Figure 5.5 shows that all random-pairing implementations have significant cost compared to the reference. However, they are still on the same order of magnitude and 800 million operations per second is currently not anywhere close to the performance reachable by a single-node database system anyway.

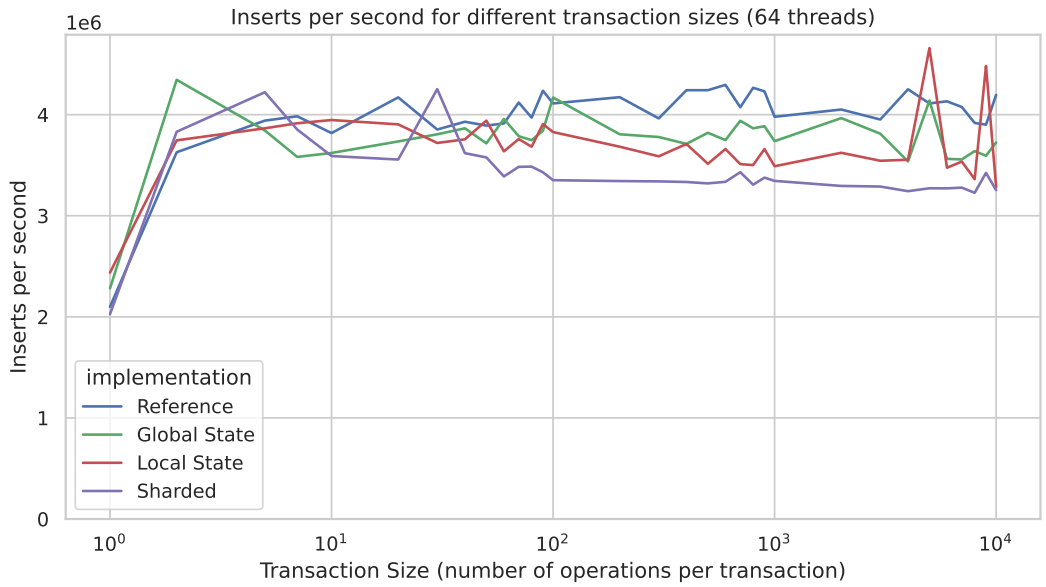


Figure 5.2.: Microbenchmark results: Inserts per second at 64 threads depending on transaction size

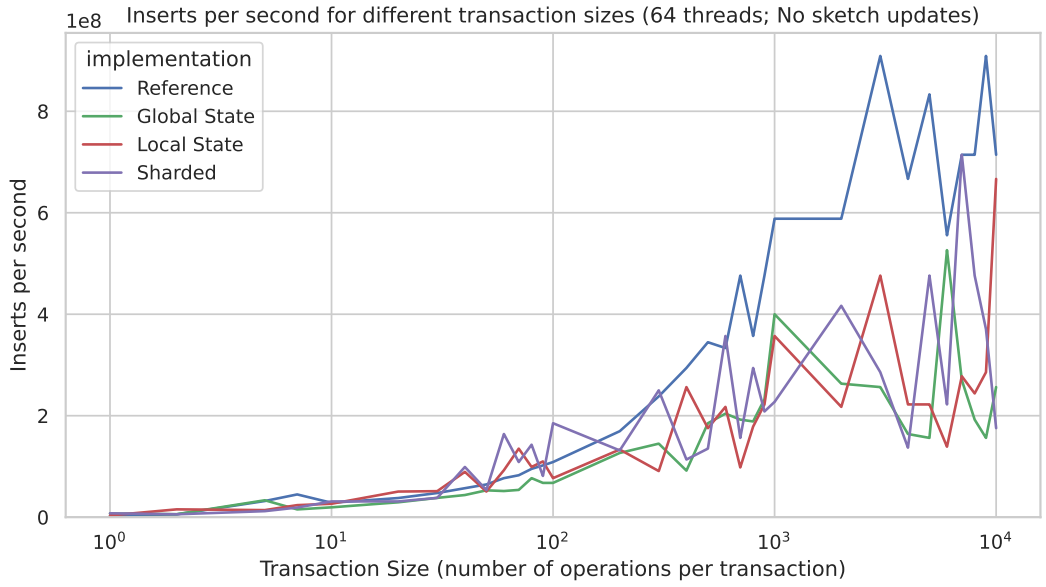


Figure 5.3.: Microbenchmark results: Inserts per second with no sketch updates at 64 threads depending on transaction size

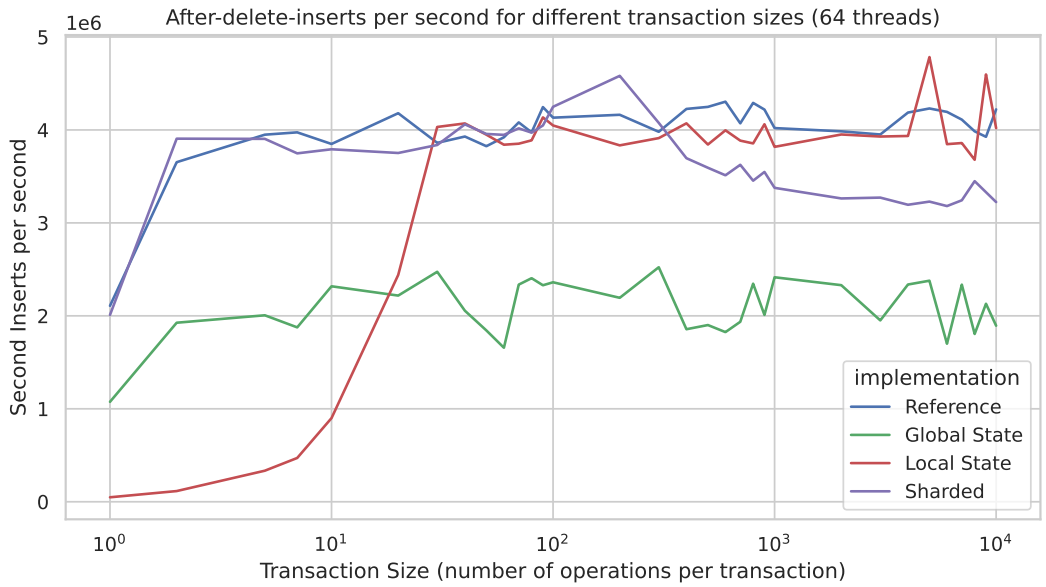


Figure 5.4.: Microbenchmark results: Inserts per second at 64 threads after previous deletes depending on transaction size

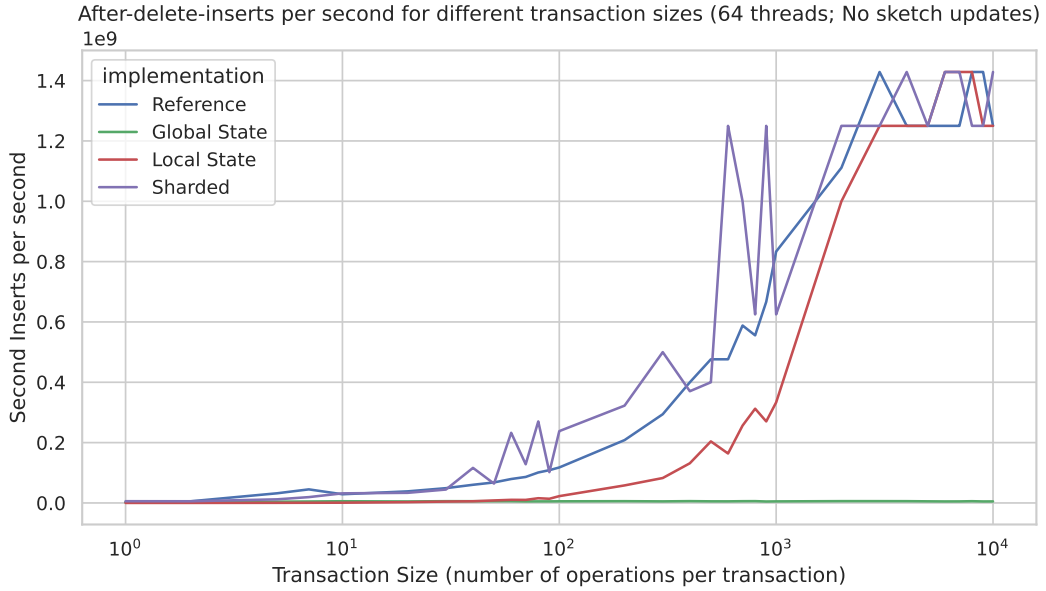


Figure 5.5.: Microbenchmark results: Inserts per second with no sketch at 64 threads after previous deletes depending on transaction size

5.4. TPC-C

For assessing the performance of an entire database system using the techniques previously described in this thesis in a transactional (OLTP) workload, TPC-C is used since it is one of the classic benchmarks for transactional systems. The benchmark is configured to use a number of client threads between 1 and 64 and is run on a dataset with 64 warehouses.

Figure 5.6 shows a comparison between the pure random-pairing based implementations. As the plot shows, the contention observed for the local-state implementation in the microbenchmark does not translate to a TPC-C workload.

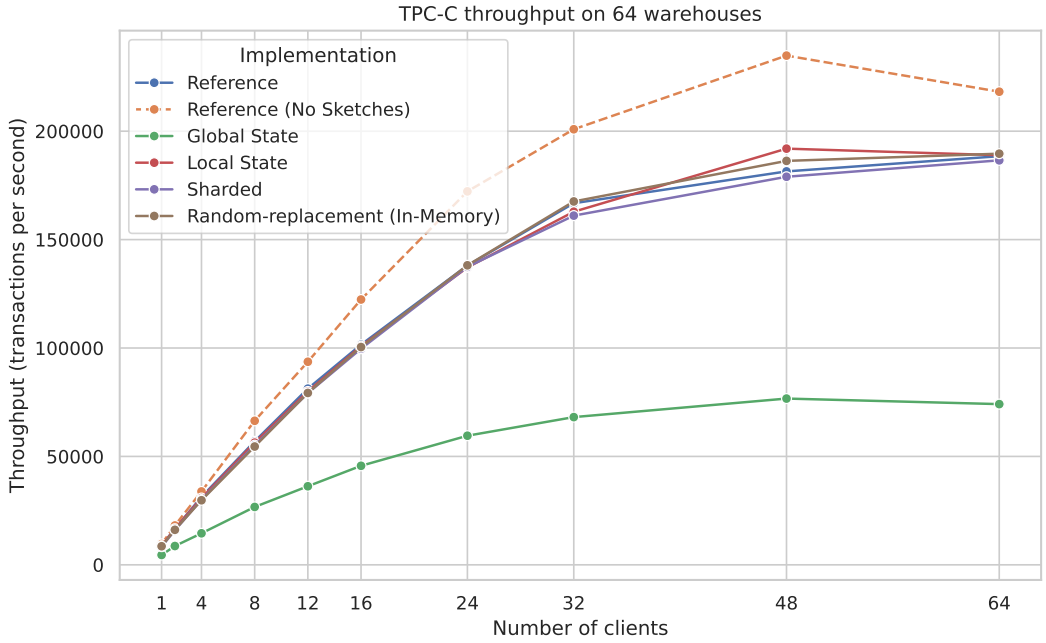


Figure 5.6.: TPC-C results for random-pairing implementation variants (64 warehouses))

These results show that the performance cost of also handling updates and deletes for online sampling is small in real systems. It is below 5% for transactional workloads. For comparison, deactivating sketch updates for inserts brings around 20% higher throughput in TPC-C. The performance of the local state and the sharding implementations are roughly identical in this case. This is probably because TPC-C does not cause a lot of contention on the random-pairing counters. There is only one table that even has deletes which is the *NEW_ORDERS* table. This table is only touched by approximately half of the transactions which reduces contention.

Most statements executed are actually UPDATE (besides SELECT) which could only cause contention on the concurrent hash table which should be more scalable than the global counters. A test that should be heavier on the counters would be one where there is just a single table and a workload that consists of just single-tuple insert/delete operations with a 50% split. Such a workload is shown in the next section.

The basic implementation with just global state has significantly lower throughput, hinting towards the thread-local state of the other implementations being beneficial even in cases where every transaction only modifies a relatively small number of tuples.

Since Umbra will generally be able to achieve at least five times more transactions per second than PostgreSQL and even more compared to existing commercial systems [FKN22], it can be assumed that even the pure global-state implementation might be good enough for almost any other OLTP system. For systems like Umbra, which achieve a higher performance and scalability in every other part of the system too, the frequent writes to global atomic counters in the basic implementation will almost halve the throughput of TPC-C transactions per second. Systems like that should definitely use a sharded implementation or at least implement local state for deletes which reduces the amount of write operations to a global state shared among all threads.

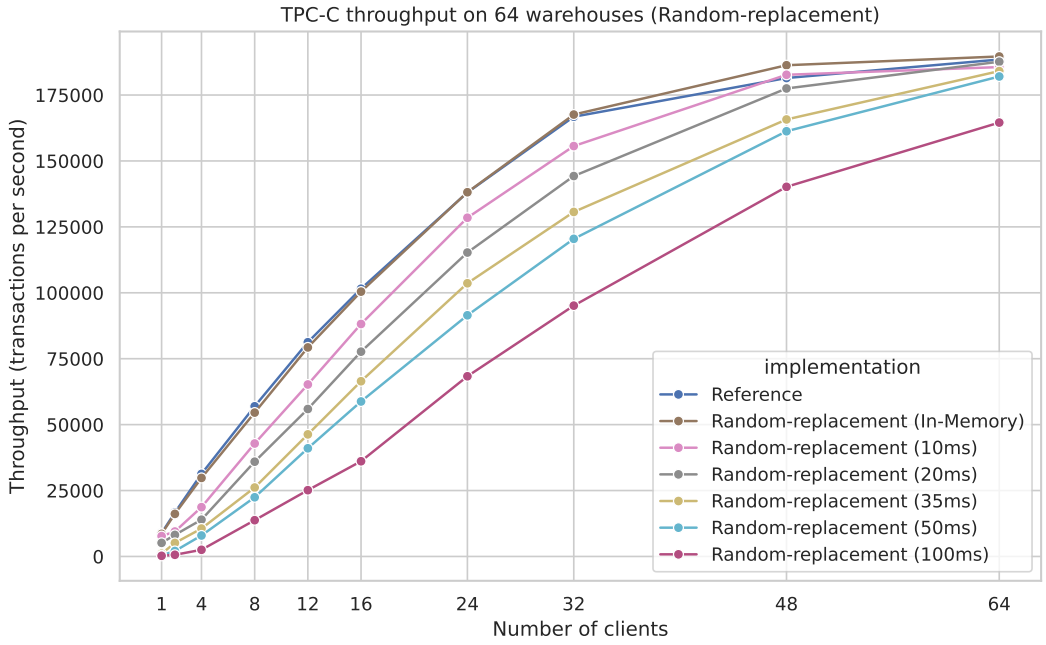


Figure 5.7.: TPC-C results for random-replacement implementation variants (64 warehouses)

Random-replacement variants are shown in Figure 5.7 with different settings for an artificial random-replacement delay to simulate wait times for I/O requests, either to disk or to a network storage service like S3. The delay is implemented as a simple sleep. The results show that for transactional workloads only in-memory systems or systems with low-latency disks like Solid State Drives (SSDs) can do random replacement without significant performance cost. Especially going through a network, possibly multiple times because of acceptance/rejection sampling, is not feasible here. Doing random replacement asynchronously could limit the effect because it does not add replacement time to the latency of the transaction.

5.5. OLTP contention benchmark

Since the previous TPC-C benchmark did not cause enough contention for the local state implementation to have any significant performance impact, another benchmark specifically designed to try to cause contention in the online statistics will be described here.

5.5.1. Benchmark design

The case that should cause the most contention on the online statistics and random-pairing data structures is one where single tuples are deleted and inserted in a one-to-one ratio since this leads to the local state optimization becoming useless.

To reduce contention in all other parts of the system, a partitioned table with two fields is used for the benchmark. It has the columns *chunk* and *id* which are also indexed in order to find single tuples quickly. The table is hash-partitioned by chunk and each client operates on a separate chunk of the table to reduce contention on the base table-storage and the index. Each client simply runs a workload that repeatedly calls a stored procedure which runs a number of delete-single-tuple and insert-single-tuple operations in its chunk.

The operations are done in a way that always deletes the tuple with the lowest id and inserts it again with the highest id. The lowest id is tracked in the benchmarking client and the new highest id can be calculated by adding the fixed cardinality. The number of operations done per procedure call will be called transaction size again and is set to 64 for the results shown here. The table is pre-filled with a fixed number of tuples which is the cardinality of the table over the entire benchmark duration. The cardinality per client (which is the cardinality per chunk) will be set to 100,000 for the results shown. The full definition of the table and the stored procedure can be found in the appendix Section A.2.

5.5.2. Results

As can be seen in Figure 5.8, which compares the pure random-pairing-based implementations, the implementation with global counters and thread-local state is basically indistinguishable from the pure global state implementation. This is the expected result since the local state optimization brings no benefit for operations that touch exactly one tuple. This is because the thread-local state will be merged into the global state after every tuple, which is almost identical to applying the operations to the global state directly. The sharded implementation, on the other hand, only has a small cost relative to the reference which is mainline Umbra with contention in local insert-state initialization, due to unnecessary atomic loading of shared pointers, removed. Random-replacement variants will not be shown here, because random replacement itself does not cause significant contention inside the online statistics.

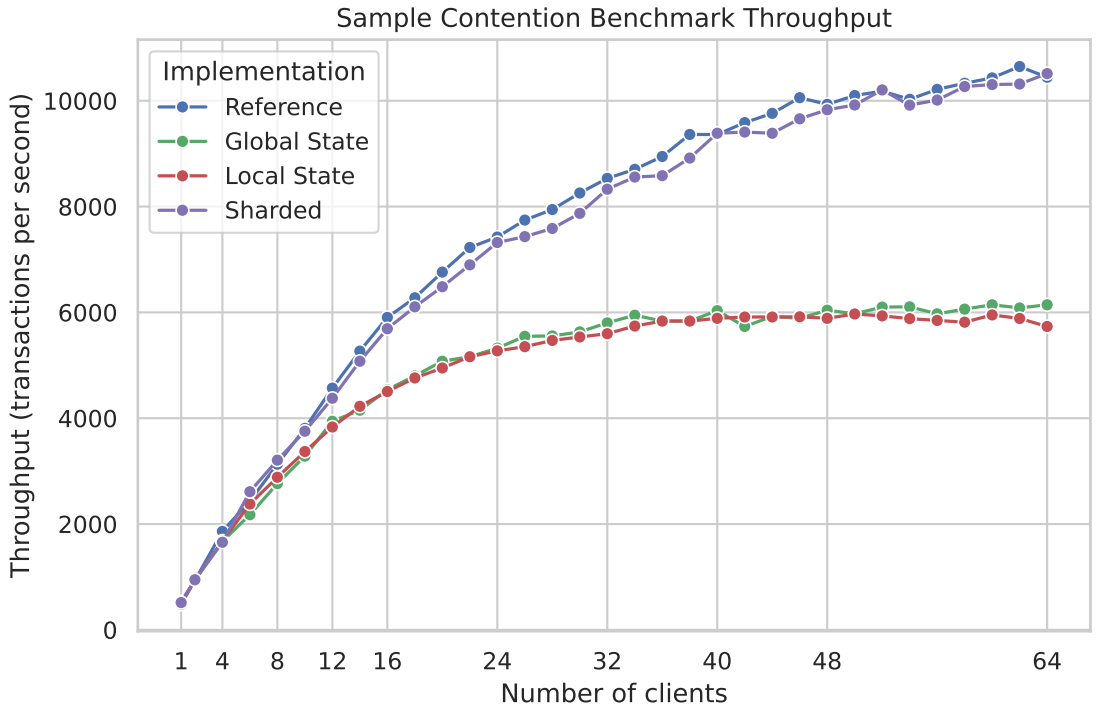


Figure 5.8.: Sample contention benchmark for implementations without random replacement

5.6. Bulk Load/Bulk Delete

For many Data Warehousing or OLAP workloads, the use case requires bulk loading either the entire dataset or periodical diffs in the form of Extract, transform, load (ETL) or Extract, load, transform (ELT) pipelines. Both of these cases mean that, usually, many thousands of tuples will be inserted at once in a small number of transactions. Two benchmarks were evaluated to simulate this. One is doing some simple bulk load and delete operations on a simple table with randomly generated data. The other is bulk operations on the largest TPC-H table, *lineitem*.

5.6.1. Simple Random Data Bulk Load/BulkDelete

A basic test table describing persons (see Listing 5.1) is created and filled with random data.

```
1 CREATE TABLE test (
2   id INT PRIMARY KEY,
3   name TEXT,
4   age INT,
5   email TEXT
6 ) WITH (storage = 'paged');
```

Listing 5.1: Bulk-Load/Bulk-Delete Benchmark: table definition

The data is randomly generated by the database system itself using the random() function. Strings are created by getting md5 hashes from a text representation of the random number using md5(random()::text). Listing 5.2 shows the full insert statement. The benchmark can be run with different table sizes, however, the sizes chosen must be large enough such that the overall runtime of both the insert as well as the delete is dominated by actual processing and not the one-time overhead of parsing or query-planning.

```
1 INSERT INTO test (id, name, age, email)
2   SELECT
3     row_number() OVER (),
4     md5(random()::text),
5     floor(random() * 100) + 1,
6     md5(random()::text) || '@example.com'
7   FROM
8     generate_series(1, 1000000);
```

Listing 5.2: Bulk-Load/Bulk-Delete Benchmark: Bulk Loading with random data

The age field is filled with uniformly distributed random values between 1 and 100. Afterward, all the values with an age less than 50 will be deleted (DELETE FROM

test **WHERE** age < 50). This should lead to roughly half of the tuples in the table being deleted. Then after that, the initial insert will be repeated with different IDs to check the performance of inserts with previous deletes. This has no impact on the reference which does not handle deletes but on the non-replacing implementations that use some kind of state (counters) to track previously deleted tuples.

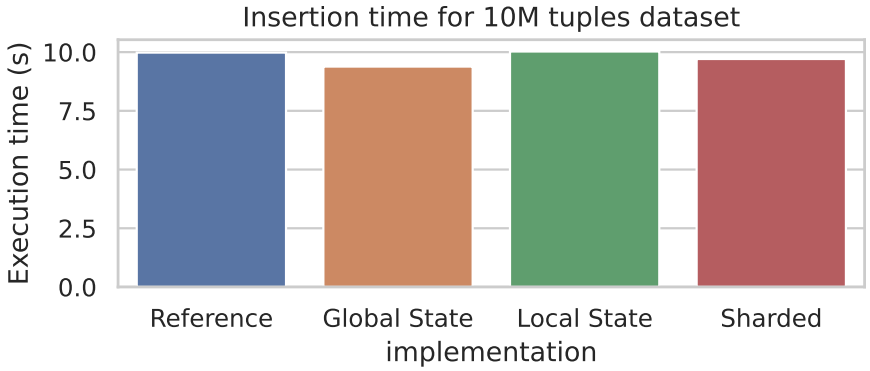


Figure 5.9.: Random dataset workload: Insert performance for 10 Million tuples

As can be seen in the plot in Figure 5.9 the insertion performance for the initial inserts is identical for all implementations. This is the expected result since no previous deletes happened which means random pairing will not be used here and the random-pairing counters will not be touched.

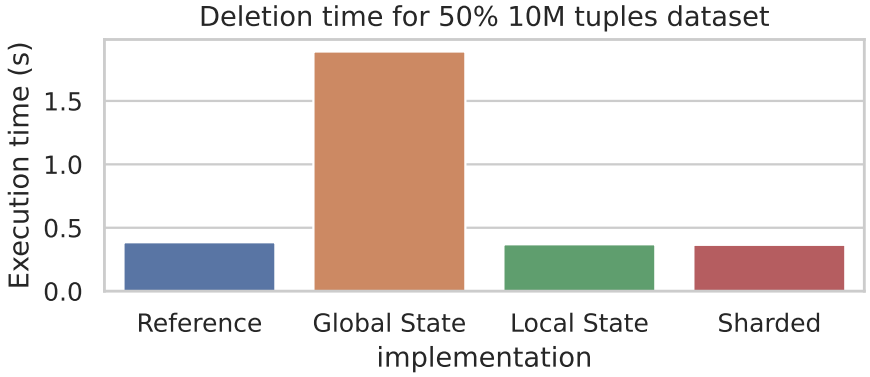


Figure 5.10.: Random dataset workload: Deletion performance for deleting 5 Million out of 10 Million tuples

For deletes, Figures 5.10 and 5.12 show that the local state plays a major role here. Without it, the performance is worse by roughly a factor of 4. Reinsertions, on the

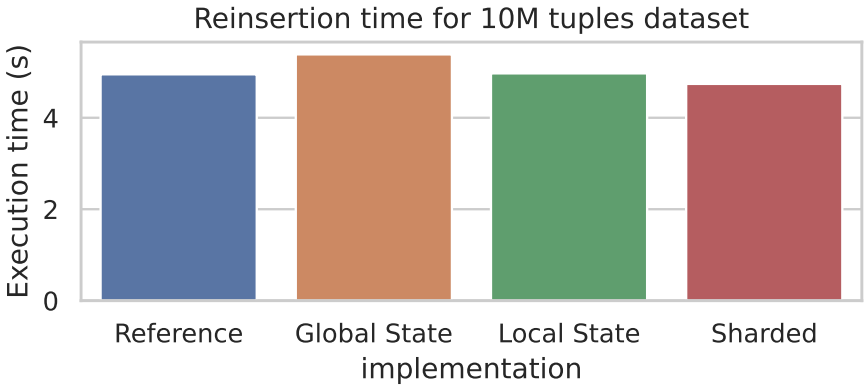


Figure 5.11.: Random dataset workload: Reinsert performance for 5 Million tuples after deleting 5 Million out of 10 Million tuples

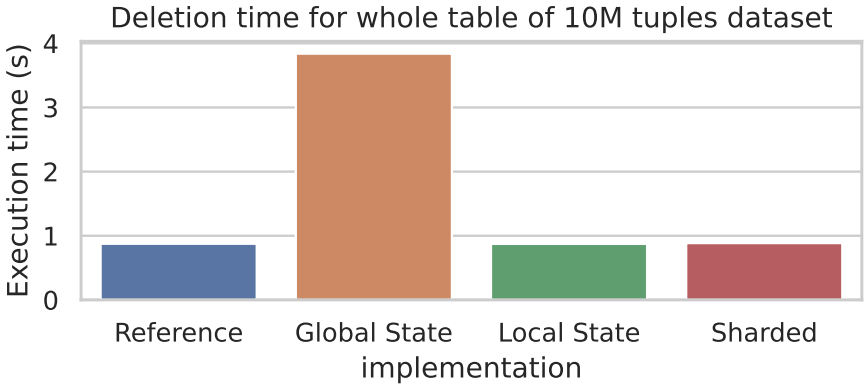


Figure 5.12.: Random dataset workload: Deletion performance for deleting the entire table of 10 Million tuples

other hand, (Figure 5.11) are only slightly slower without the local state. This is likely because, for insertions, more other things need to be done which can also be seen in the total execution time for the reference variant being roughly 10 times longer. For deletes, the tuple is only marked as deleted whereas for inserts, the physical insert to a page needs to happen. Therefore, the overall contention will also be much lower and be a smaller factor in total execution time.

5.6.2. TPC-H bulk operations

The second test is perhaps more close to real-world use cases. It uses the largest table in the TPC-H benchmark, which is one of the classic benchmarks for analytical workloads. Scale factor 10 was chosen to minimize the impact of constant query processing overhead. The table is loaded into memory from a CSV, then some full table scans (including TPC-H query 1 and 6) are issued to warm up caches, and then a copy statement (`CREATE TABLE lineitem2 AS (SELECT * FROM lineitem)`) is issued after which all rows of the table are deleted (`DELETE FROM lineitem`).

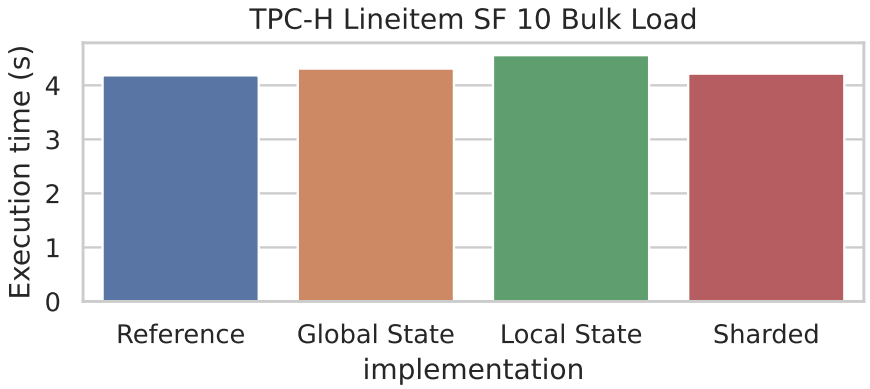


Figure 5.13.: TPC-H bulk load benchmark for random-pairing variants

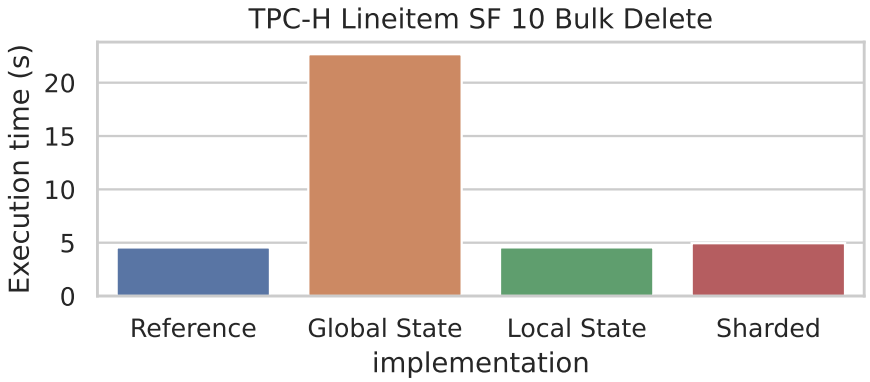


Figure 5.14.: TPC-H bulk delete benchmark for random-pairing variants

Figures 5.13 and 5.14 show similar results to the ones shown in the previous section for similar reasons.

5.7. OddSketch accuracy

To show how accurate OddSketches [MPP14] are for estimating set differences, a similar configuration to the one described in the last chapter for Umbra was implemented in Python. WyHash [Yi] is used for a hash function and two random 1024-element sets with a configurable amount of common elements are inserted into OddSketches of 512 bits which are then used for difference estimation.

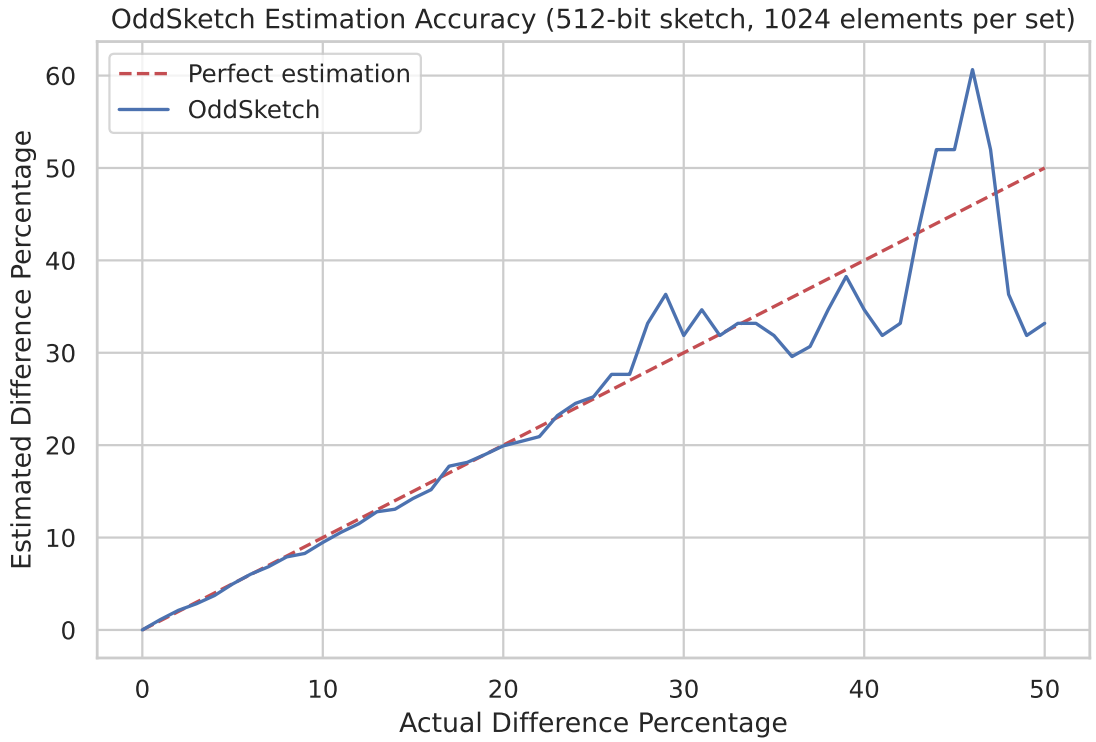


Figure 5.15.: Accuracy of a 512-bit OddSketch for random 1024-element sets

Figure 5.15 shows the accuracy of the OddSketch is very high until around 25% difference which is more than enough for the use case described in the previous chapter.

6. Discussion

This thesis showed that through a scalable implementation of random pairing, updates, and deletes can be built into online sampling on top of existing reservoir sampling algorithms. It also shows that this can be done with a very small performance impact in both transactional- and bulk workloads. A system can choose to implement just the variant with thread-local state but without sharding the global state if the expected usage is bulk workloads. When a lot of very small transactions are within the expected use, a sharded implementation is necessary to not limit the performance of the system through statistics maintenance.

If a compressed representation for the sample is used for cardinality estimation, OddSketches can be used to quickly but accurately determine when the online sample has to be recompressed.

6.1. Future Work

6.1.1. Making other statistics updatable

This thesis only touched on the sampling part of Umbra's statistics collection. However, the other components of statistics, namely HyperLogLog and Fast-AMGS sketches, should eventually also have updates and deletes on them enabled. Both should be rather trivial to implement but might come with some performance costs. For HyperLogLog sketches, it has already been shown that through the use of *Couting HyperLogLog sketches*, they can be made to handle deletes. When using the inexact variant with probabilistic counters, the memory overhead is also manageable [FN19]. Fast-AMGS sketches are already counting sketches, so the counters can just be decreased for deletes. Updates are then handled like a delete followed by an update, at least for the updated columns.

It might also be useful to include other kinds of sketches. The Multi-resolution OddSketches mentioned previously [XWZ21] or some new sketch technique developed on top of them might be a good fit for estimating the selectivity of joins, for example.

6.1.2. Reusing existing table scans for filling up the sample

A solution not considered in our evaluation that could help, especially for degenerate cases, would be to piggyback off full table scans and resample either the entire sample or at least the missing part of it. Changes to the online statistics from incoming transactions would have to be redirected or multiplexed to the new online statistics instance before starting the resampling. Otherwise, the operations executed during the statistics recomputation might be lost. If the old online statistics instance is replaced immediately, compression and serialization of the online statistics into compressed statistics must also be blocked until the recomputation is done.

A. Appendix

A.1. OddSketch estimation procedure Assembly

In the following Appendix, the assembly output for different configurations of GCC 13.2 and Clang 17.0.1 for the xor-popcount section of the difference estimation function of OddSketches will be shown.

A.1.1. Zen4

GCC

This is the example already previously shown and is just added here for completeness.

```
xorpopcnt(std::array<unsigned long, 8ul>&, std::array<unsigned long, 8ul>&):
  vmovdqu64 zmm0, ZMMWORD PTR [rdi]
  vpxorq zmm0, zmm0, ZMMWORD PTR [rsi]
  vpopcntq zmm0, zmm0
  vextracti64x4 ymm1, zmm0, 0x1
  vpaddq ymm1, ymm1, ymm0
  vmovdqa xmm0, xmm1
  vextracti64x2 xmm1, ymm1, 0x1
  vpaddq xmm0, xmm0, xmm1
  vpsrldq xmm1, xmm0, 8
  vpaddq xmm0, xmm0, xmm1
  vmovq rax, xmm0
  vzeroupper
  ret
```

Listing A.1: Assembly generated by GCC for counting the 1-bits in the result of xoring two OddSketches with 64-byte length on Zen4

Clang

Clang produces shorter code which utilizes the vpsadbw (Compute Sum of Absolute Differences) instruction for the horizontal-add part.

```
xorpopcnt(std::array<unsigned long, 8ul>&, std::array<unsigned long, 8ul>&):
  vmovdqu64 zmm0, zmmword ptr [rsi]
  vpxor xmm1, xmm1, xmm1
  vpxorq zmm0, zmm0, zmmword ptr [rdi]
  vpopcntq zmm0, zmm0
  vpmovqb xmm0, zmm0
  vpsadbw xmm0, xmm0, xmm1
  vmovq rax, xmm0
  vzeroupper
  ret
```

Listing A.2: Assembly generated by Clang for counting the 1-bits in the result of xoring two OddSketches with 64-byte length on Zen4

A.1.2. IceLake (server)

For Intel processors similarly short results are achieved.

GCC

GCC produces a result very similar to the one it produces for the AMD Zen4 architecture option.

```
xorpopcnt(std::array<unsigned long, 8ul>&, std::array<unsigned long, 8ul>&):
  vmovdqu64 zmm0, ZMMWORD PTR [rdi]
  vpxorq zmm0, zmm0, ZMMWORD PTR [rsi]
  vpopcntq zmm0, zmm0
  vextracti64x4 ymm1, zmm0, 0x1
  vpaddq ymm1, ymm1, ymm0
  vmovdqa xmm0, xmm1
  vextracti64x2 xmm1, ymm1, 0x1
  vpaddq xmm0, xmm0, xmm1
  vpsrldq xmm1, xmm0, 8
  vpaddq xmm0, xmm0, xmm1
  vmovq rax, xmm0
  vzeroupper
  ret
```

Listing A.3: Assembly generated by GCC for counting the 1-bits in the result of xoring two OddSketches with 64-byte length on Icelake

Clang

Clang interestingly produces code that uses 2 256-bit vectors instead of a single 512-bit vector.

```
xorpopcnt(std::array<unsigned long, 8ul>&, std::array<unsigned long, 8ul>&):
vmovdqu ymm0, ymmword ptr [rsi]
vmovdqu ymm1, ymmword ptr [rsi + 32]
vpxor ymm1, ymm1, ymmword ptr [rdi + 32]
vpxor ymm0, ymm0, ymmword ptr [rdi]
vpopcntq ymm0, ymm0
vpopcntq ymm1, ymm1
vpmovqb xmm1, ymm1
vpmovqb xmm0, ymm0
vpunpckldq xmm0, xmm0, xmm1 # xmm0 = xmm0[0],xmm1[0],xmm0[1],xmm1[1]
vpxor xmm1, xmm1, xmm1
vpsadbw xmm0, xmm0, xmm1
vmovq rax, xmm0
vzeroupper
ret
```

Listing A.4: Assembly generated by Clang for counting the 1-bits in the result of xoring two OddSketches with 64-byte length on Icelake

A.1.3. Skylake

Skylake is an example of a modern but somewhat older architecture with AVX-2 but without AVX-512 support. What can be seen here is that Clang/LLVM optimizes more aggressively for SIMD whereas GCC falls back to rather straightforward translation with an unrolled loop which will likely still have quite good performance because of superscalar execution. This is also similar to the code generated by both compilers for even older processor architectures like nehalem.

GCC

```
xorpopcnt(std::array<unsigned long, 8ul>&, std::array<unsigned long, 8ul>&):
  mov rdx, rsi
  mov rcx, QWORD PTR [rdi+8]
  mov rsi, QWORD PTR [rsi]
  mov rax, QWORD PTR [rdi+56]
  xor rsi, QWORD PTR [rdi]
  xor rcx, QWORD PTR [rdx+8]
  popcnt rcx, rcx
  popcnt rsi, rsi
  add rsi, rcx
  mov rcx, QWORD PTR [rdi+16]
  xor rax, QWORD PTR [rdx+56]
  xor rcx, QWORD PTR [rdx+16]
  popcnt rax, rax
```

```
    popcnt rcx, rcx
    add rcx, rsi
    mov rsi, QWORD PTR [rdi+24]
    xor rsi, QWORD PTR [rdx+24]
    popcnt rsi, rsi
    add rsi, rcx
    mov rcx, QWORD PTR [rdi+32]
    xor rcx, QWORD PTR [rdx+32]
    popcnt rcx, rcx
    add rcx, rsi
    mov rsi, QWORD PTR [rdi+40]
    xor rsi, QWORD PTR [rdx+40]
    popcnt rsi, rsi
    add rsi, rcx
    mov rcx, QWORD PTR [rdi+48]
    xor rcx, QWORD PTR [rdx+48]
    popcnt rcx, rcx
    add rcx, rsi
    add rax, rcx
    ret
```

Listing A.5: Assembly generated by GCC for counting the 1-bits in the result of xoring two OddSketches with 64-byte length on Skylake

Clang

```
.LCPI0_2:
    .byte 15 # 0xf
.LCPI0_3:
    .byte 0 # 0x0
    .byte 1 # 0x1
    .byte 1 # 0x1
    .byte 2 # 0x2
    .byte 1 # 0x1
    .byte 2 # 0x2
    .byte 2 # 0x2
    .byte 3 # 0x3
    .byte 1 # 0x1
    .byte 2 # 0x2
    .byte 2 # 0x2
    .byte 3 # 0x3
    .byte 2 # 0x2
    .byte 3 # 0x3
    .byte 3 # 0x3
    .byte 4 # 0x4
xorpopcnt(std::array<unsigned long, 8ul>&, std::array<unsigned long, 8ul>&):
```

```
21  vmovdqu ymm0, ymmword ptr [rsi]
22  vmovdqu ymm1, ymmword ptr [rsi + 32]
23  vpxor ymm0, ymm0, ymmword ptr [rdi]
24  vpxor ymm1, ymm1, ymmword ptr [rdi + 32]
25  vpbroadcastb ymm2, byte ptr [rip + .LCPI0_2] # ymm2 =
        [15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15]
26  vpand ymm3, ymm1, ymm2
27  vbroadcasti128 ymm4, xmmword ptr [rip + .LCPI0_3] # ymm4 =
        [0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4]
28  vpshufb ymm3, ymm4, ymm3
29  vpsrlw ymm1, ymm1, 4
30  vpand ymm1, ymm1, ymm2
31  vpshufb ymm1, ymm4, ymm1
32  vpaddb ymm1, ymm1, ymm3
33  vpxor xmm3, xmm3, xmm3
34  vpsadbw ymm1, ymm1, ymm3
35  vpand ymm5, ymm0, ymm2
36  vpshufb ymm5, ymm4, ymm5
37  vpsrlw ymm0, ymm0, 4
38  vpand ymm0, ymm0, ymm2
39  vpshufb ymm0, ymm4, ymm0
40  vpaddb ymm0, ymm0, ymm5
41  vpsadbw ymm0, ymm0, ymm3
42  vpaddq ymm0, ymm0, ymm1
43  vextracti128 xmm1, ymm0, 1
44  vpaddq xmm0, xmm0, xmm1
45  vpshufd xmm1, xmm0, 238 # xmm1 = xmm0[2,3,2,3]
46  vpaddq xmm0, xmm0, xmm1
47  vmovq rax, xmm0
48  vzeroupper
49  ret
```

Listing A.6: Assembly generated by Clang for counting the 1-bits in the result of xoring two OddSketches with 64-byte length on Skylake

A.2. Sample Contention Benchmark definition

A.2.1. Table definition

```
CREATE TABLE benchmark
(
chunk INTEGER NOT NULL,
id INTEGER NOT NULL
) PARTITION BY HASH (chunk) WITH (storage = paged);

CREATE INDEX ON benchmark (chunk, id);
```

Listing A.7: DDL for creating the sample contention benchmark schema

A.2.2. Workload procedure

```
CREATE PROCEDURE delete_insert(var_chunk INTEGER, var_id_start INTEGER, var_id_end
      INTEGER, var_card INTEGER)
AS
$$
SELECT id AS var_id FROM generate_series(var_id_start, var_id_end) g(id) {
  DELETE
  FROM benchmark
  WHERE chunk = var_chunk and id = var_id
  catch serialization_failure {
    raise error 'serialization failure';
    return;
  };

  INSERT INTO benchmark (chunk, id)
  VALUES (var_chunk, var_id + var_card)
  catch serialization_failure {
    raise error 'serialization failure';
    return;
  };
}

COMMIT;
$$ LANGUAGE 'umbrascript';
```

Listing A.8: UmbraScript definition for the procedure representing the workload of the sample contention benchmark

Abbreviations

TID Tuple ID

OLTP Online transaction processing

OLAP Online analytical processing

HTAP Hybrid transactional/analytical processing

ETL Extract, transform, load

ELT Extract, load, transform

MVCC Multi-Version Concurrency Control

PDF Probability density function

LSN Log Sequence Number

SSD Solid State Drive

List of Figures

List of Listings

Bibliography

[AD52] T. W. Anderson and D. A. Darling. “Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes.” In: *The Annals of Mathematical Statistics* 23.2 (1952). Publisher: Institute of Mathematical Statistics, pp. 193–212. ISSN: 0003-4851.

[App] A. Appleby. *MurmurHash3*. `https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp`. [Accessed 08-03-2024].

[AT06] S. Agarwal and A. Trachtenberg. “Approximating the number of differences between remote sets.” In: *2006 IEEE Information Theory Workshop - ITW '06 Punta del Este*. 2006 IEEE Information Theory Workshop - ITW '06 Punta del Este. Mar. 2006, pp. 217–221. DOI: 10.1109/ITW.2006.1633815.

[Boo] Boost Organization. *Boost C++ libraries*. `https://www.boost.org/`. [Accessed 08-03-2024].

[BRN20] A. Birler, B. Radke, and T. Neumann. “Concurrent Online Sampling for All, for Free.” In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. DaMoN '20. Portland, Oregon: Association for Computing Machinery, 2020. ISBN: 9781450380249. DOI: 10.1145/3399666.3399924.

[CG05] G. Cormode and M. Garofalakis. “Sketching streams through the net: distributed approximate query tracking.” In: *Proceedings of the 31st international conference on Very large data bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, Aug. 30, 2005, pp. 13–24. ISBN: 978-1-59593-154-2.

[CLM85] P. Celis, P.-A. Larson, and J. I. Munro. “Robin hood hashing.” In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 1985, pp. 281–288. DOI: 10.1109/SFCS.1985.48.

[FKN22] M. Freitag, A. Kemper, and T. Neumann. “Memory-optimized multi-version concurrency control for disk-based database systems.” In: *Proc. VLDB Endow.* 15.11 (July 2022), pp. 2797–2810. ISSN: 2150-8097. DOI: 10.14778/3551793.3551832.

[Fla+07] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm." In: *Discrete mathematics & theoretical computer science* Proceedings (2007).

[FN19] M. J. Freitag and T. Neumann. "Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates." In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[Fra04] K. Fraser. "Practical lock-freedom." PhD thesis. University of Cambridge, 2004.

[GL13] D. Guo and M. Li. "Set Reconciliation via Counting Bloom Filters." In: *IEEE Transactions on Knowledge and Data Engineering* 25.10 (2013), pp. 2367–2380. DOI: 10.1109/TKDE.2012.215.

[GLH06] R. Gemulla, W. Lehner, and P. J. Haas. "A Dip in the Reservoir: Maintaining Sample Synopses of Evolving Datasets." In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 595–606.

[HNH13] S. Heule, M. Nunkesser, and A. Hall. "HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm." In: *Proceedings of the 16th International Conference on Extending Database Technology*. EDBT '13. Genoa, Italy: Association for Computing Machinery, 2013, pp. 683–692. ISBN: 9781450315975. DOI: 10.1145/2452376.2452456.

[KLN21] T. Kersten, V. Leis, and T. Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra." In: *The VLDB Journal* 30.5 (June 2021), pp. 883–905. ISSN: 1066-8888. DOI: 10.1007/s00778-020-00643-4.

[KPM18] R. Kelly, B. A. Pearlmutter, and P. Maguire. "Concurrent Robin Hood Hashing." In: *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Ed. by J. Cao, F. Ellen, L. Rodrigues, and B. Ferreira. Vol. 125. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 10:1–10:16. ISBN: 978-3-95977-098-9. DOI: 10.4230/LIPIcs.OPODIS.2018.10.

[Lei+14] V. Leis, P. Boncz, A. Kemper, and T. Neumann. "Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 743–754. ISBN: 9781450323765. DOI: 10.1145/2588555.2610507.

[Lei+15] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. "How good are query optimizers, really?" In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 204–215. ISSN: 2150-8097. DOI: 10.14778/2850583.2850594.

[Li94] K.-H. Li. "Reservoir-sampling algorithms of time complexity O(n(1 + log(N/n)))." In: *ACM Transactions on Mathematical Software* 20.4 (Dec. 1, 1994), pp. 481–493. ISSN: 0098-3500. DOI: 10.1145/198429.198435.

[MPP14] M. Mitzenmacher, R. Pagh, and N. Pham. "Efficient estimation for high similarities using odd sketches." In: *Proceedings of the 23rd international conference on World wide web*. WWW '14: 23rd International World Wide Web Conference. Seoul Korea: ACM, Apr. 7, 2014, pp. 109–118. ISBN: 978-1-4503-2744-2. DOI: 10.1145/2566486.2568017.

[Neu11] T. Neumann. "Efficiently compiling efficient query plans for modern hardware." In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940.

[NF20] T. Neumann and M. J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[OR89] F. Olken and D. Rotem. "Random sampling from B+ trees." In: *Proceedings of the 15th International Conference on Very Large Data Bases*. VLDB '89. Amsterdam, The Netherlands: Morgan Kaufmann Publishers Inc., 1989, pp. 269–277. ISBN: 1558601015.

[TW11] S. Tirthapura and D. P. Woodruff. "Optimal Random Sampling from Distributed Streams Revisited." In: *Proceedings of the 25th International Conference on Distributed Computing*. DISC'11. Rome, Italy: Springer-Verlag, 2011, pp. 283–297. ISBN: 9783642240997.

[Vit85] J. S. Vitter. "Random Sampling with a Reservoir." In: *ACM Trans. Math. Softw.* 11.1 (Mar. 1985), pp. 37–57. ISSN: 0098-3500. DOI: 10.1145/3147.3165.

[WKN21] B. Wagner, A. Kohn, and T. Neumann. "Self-Tuning Query Scheduling for Analytical Workloads." In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 1879–1891. ISBN: 9781450383431. DOI: 10.1145/3448016.3457260.

[XWZ21] Q. Xiao, L. Wen, and Q. Zhang. "Multi-resolution Odd Sketch for Mining Jaccard Similarities between Dynamic Streaming Sets." In: *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. 2021, pp. 198–203. DOI: 10.1109/CSCWD49262.2021.9437641.

[Yi] W. Yi. *WyHash*. https://github.com/wangyi-fudan/wyhash. [Accessed 08-03-2024].