Demonstrating Transfer-Efficient Sample Maintenance on Graphics Cards

Martin Kiefer
Technische Universität Berlin, Germany
kiefer@campus.tu-berlin.de

Max Heimel
Technische Universität Berlin, Germany
max.heimel@tu-berlin.de

Volker Markl
Technische Universität Berlin, Germany
volker.markl@tu-berlin.de

ABSTRACT

Maintaining random data samples under database updates is a fundamental operation in modern database engines. While multiple algorithms exist for this problem, none is tailored to the special case of maintaining data samples on graphics cards. Due to the limited interconnect bandwidth to main memory, any GPU-resident algorithm must try to avoid data transfers across the PCI Express bus where possible – a property that we call transfer-efficient. In this demonstration, we present an approximate, transfer-efficient sample maintenance algorithm that piggybacks on a GPU-accelerated selectivity estimator and utilizes query feedback to selectively identify and replace outdated points. We provide an implementation of the algorithm and interactively demonstrate its quality and its transfer performance in comparison to traditional maintenance algorithms.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

1. INTRODUCTION & MOTIVATION

Collecting and maintaining data samples is a fundamental operation in a modern database engine. One of the main advantages of algorithms that can operate on samples lies in their ability to trade-off performance against result accuracy by reducing – or increasing – the sample size. This property allows us to efficiently mask limited resources, as long as our application can tolerate the loss of result accuracy. Examples of such "tolerant" applications include selectivity estimation [14, 15], approximate query processing [3, 4], data mining algorithms [17], interactive data exploration, and data visualization.

One area where sampling-based methods are of particular interest are GPU-accelerated databases. The usability of graphics cards for data-intensive operations is severely limited by two bottlenecks: The scarce availability of on-card device memory and the slow data transfer speeds from main memory across the PCI Express bus [6]. We can avoid both bottlenecks by keeping a fixed number of sampled data points on the device to quickly compute approximate results from. As long as the underlying database remains static, this approach works very well and does not require any further transfers across the PCI Express bus.

Sadly, in the real world, datasets seldomly remain static. In order to stay representative, all changes that are applied to the underlying database have to be mirrored to the sample as well. This so-called *sample maintenance* problem is well-known, and multiple algorithms exist for it [8, 9, 16, 18]. However, while maintenance algorithms guarantee to keep the sample representative, they usually require us to replay database updates. In case of a GPU-resident data sample, this means that we need to copy all updates across the PCI Express bus. These additional transfers restrict the available PCI Express bandwidth, leading to performance penalties for "actual" data processing applications running on the GPU. Ideally, we want a *transfer-efficient* maintenance algorithm that only transfers data if it is absolutely necessary. In this paper, we are discussing a possible sample maintenance algorithm that aims for this property. Our primary contributions are:

1. We introduce an approximate, but transfer-efficient maintenance algorithm for samples on graphics cards. Our algorithm piggybacks on a GPU-accelerated selectivity estimator and utilizes query feedback to track outdated points. This approach allows us to selectively replace outdated points in the sample without having to mirror all updates to the graphics card.

2. We provide an implementation of our algorithm integrated into the open-source relational database engine PostgreSQL[1].

3. We interactively demonstrate the performance of our algorithm in comparison to other replacement strategies with regard to both sample quality and the required data transfers across the PCI Express bus.

2. GPU-BASED SAMPLE MAINTENANCE

Assume a d-dimensional relation R with cardinality $|R|$ that is stored within a "regular" relational database system. From R, we collect a fixed-size random sample $S \subseteq R$ and push it to the graphics card. The sample size $|S|$ is fixed and chosen in advance to a) provide sufficient confidence for the

[1]The source code is available at: `goo.gl/aQSQNd`.

10.5441/002/edbt.2015.46

approximated results, and to b) fit within the limited device memory on the graphics card. Maintaining such a GPU-resident sample when database updates occur on the host is an interesting problem that, to the best of our knowledge, has not been discussed in the literature so far.

The simplest maintenance scenario is an insertion-only workload. In this case, *Reservoir Sampling* [18] is the ideal choice: It pushes newly inserted points to the sample with probability $|S| / |R|$, replacing a random sample point. From a transfer-efficiency perspective, Reservoir Sampling is optimal: We only push exactly those data items to the graphics card that end up in the sample.

The general case of mixed workloads containing insertions, updates and deletions is more interesting: General maintenance algorithms usually handle insertions similar to Reservoir Sampling, but have special rules to deal with updates and deletions [8, 9, 16]. Take for example the *Correlated Acceptance Rejection Sampling algorithm* (CAR) [16]: When a new point $\vec{t}$ is inserted into R, a random number n is drawn from the binomial distribution $BINOM(|S|, 1/|R|)$ and n random sample points are replaced by an instance of $\vec{t}$. When a deletion occurs, all instances of the deleted point are removed from the sample and exchanged by points drawn uniformly from R. Updates are handled by directly applying them on the sample. This means that, while insertions incur maintenance costs of $\mathcal{O}(1)$, deletions and updates require additional costs of $\mathcal{O}(|S|)$.

A straightforward way to adjust algorithms like CAR for GPU-resident samples would be to maintain a sample copy on the host, apply the maintenance algorithm there, and then mirror sample updates to the GPU memory. While this approach would indeed be transfer-efficient, we still had to pay the $\mathcal{O}(|S|)$ maintenance costs for every update and deletion. For large sample sizes, these additional costs can become quite significant and slow down the system, which is why we want to push as much of the maintenance work as possible directly to the faster graphics card.

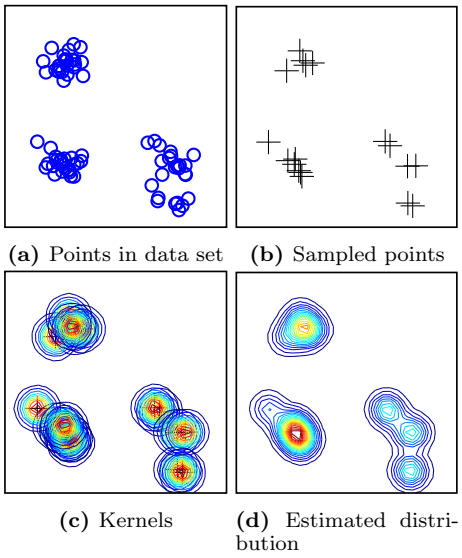
However, running existing sample maintenance algorithms on the graphics card requires us to mirror every deletion and update across the PCI Express bus, even if they do not apply to any points in the sample. For instance: Running CAR on the graphics cards incurs a sequence of two mandatory transfers for each deletion: One to transfer the deleted item, and one to reply with a list – or bitmap – identifying all deleted tuples, so that the database can sample a sufficient amount of tuples and transfer them to the correct positions in the GPU memory. These additional transfers across the limited PCI Express bus might easily become a problem: Even if they do not fully block the bus, they will take a significant chunk of the available bandwidth away from other GPU-resident applications. This is especially relevant when keeping in mind that transfers below a minimum length (on the order of a few Kilobytes) do not achieve maximum throughput [7].

3. BACKGROUND: CALCULATING KERNEL DENSITY ESTIMATORS ON GPUS

We investigated the sample maintenance problem in the context of a GPU-accelerated selectivity estimator [11]. In order to convey the necessary background knowledge, we will now give a brief introduction into this topic.

Given a relation R with attributes $(A_1, ..., A_d)$ and an arbitrary query region $\Omega \subseteq D_1 \times ... \times D_d$, selectivity estimators approximate the fraction $\frac{|\sigma_{\vec{x} \in \Omega}(R)|}{|R|}$ of tuples qualifying the query. In our case, we assume that the attributes are from the domain of real numbers.

Multiple authors have proposed using *Kernel Density Estimators* (KDEs) to approach this task [5, 10, 11]. The principle idea behind KDEs is visualized in Figure 1: Based on a sample (Figure 1b) drawn from R (Figure 1a), KDE places local probability density functions – the so-called *kernels* – around the sample points (Figure 1c). The probability density function for the overall data is then estimated by summing and averaging over those kernels (Figure 1d).



(a) Points in data set **(b)** Sampled points

(c) Kernels **(d)** Estimated distribution

Figure 1: A Kernel Density Estimator approximates the underlying distribution of a given dataset (a) by picking a random sample (b), centering local probability distributions (kernels) around them (c), and averaging the local distributions (d).

Formally, given a sample $S = \{\vec{t}^{(1)}, \vec{t}^{(2)}, ..., \vec{t}^{(s)}\} \subseteq R$, the Kernel density Estimator $\hat{p}_H(\vec{x}) : \mathbb{R}^d \to \mathbb{R}$ is defined as:

$$\begin{aligned} \hat{p}_H(\vec{x}) &= \frac{1}{s} \sum_{i=1}^{s} K_H(\vec{t}^{(i)} - \vec{x}) \\ &= \frac{1}{s \cdot |H|} \sum_{i=1}^{s} K(H^{-1}[\vec{t}^{(i)} - \vec{x}]) \end{aligned} \quad (1)$$

Here, $K : \mathbb{R}^d \to \mathbb{R}$ denotes the *kernel function*, which defines the shape of the local probability density functions. Typical choices are Gaussian – a multivariate standard normal distribution –, or Epanechnikov, which is a truncated second-degree polynomial. $H \in \mathbb{R}^{d \times d}$ is the *bandwidth matrix*, which controls the spread of the kernel function. Picking the optimal bandwidth is a difficult problem that is out of the scope of this demonstration. We assume that it is selected by a data-driven bandwidth optimizer [1].

We can now predict the selectivity for a (rectangular) query region Ω by integrating $\hat{p}_H(\vec{x})$ over all points in the region:

$$\hat{p}_H(\Omega) = \int_{\Omega} \hat{p}_H(\vec{x}) d\vec{x} = \frac{1}{s} \sum_{i=1}^{s} \underbrace{\int_{\Omega} \frac{K(H^{-1}[\vec{t}^{(i)} - \vec{x}])}{|H|}}_{\hat{p}_H^{(i)}(\Omega)} \quad (2)$$

This equation can be efficiently evaluated in parallel: First, each thread independently computes the individual probability contribution $\hat{p}_H^{(i)}(\Omega)$ for a single sample point $\vec{t}^{(i)}$. The estimate is then computed as the averaged sum over all individual contributions – which can be efficiently computed via a parallelized binary reduction scheme [13]. For further details on KDEs, and on how we designed a GPU-accelerated selectivity estimator based on them, we kindly refer to our publication [11].

4. INTRODUCING THE KARMA METRIC

We now introduce a novel approach for sample maintenance in the context of a GPU-based Kernel Density Estimator that is used for selectivity estimation. Our approach is based on *query feedback*: After the execution of a query covering region Ω, the true selectivity $p(\Omega)$ of the region is known. The principle idea behind query feedback methods is to utilize this additional information to incrementally adjust the estimation model [2].

In particular, we can compute for each sample point $\vec{t}^{(i)}$ the impact of its probability contribution $\hat{p}_H^{(i)}(\Omega)$ on the (absolute) estimation error $\mathcal{L}_{abs}\left(p\left(\Omega\right), \hat{p}_H\left(\Omega\right)\right)$. For this, we first calculate the adjusted estimate $\hat{p}_H^{-(i)}\left(\Omega\right)$ by removing the point's contribution from the estimate $\hat{p}_H(\Omega)$:

$$\hat{p}_H^{-(i)}(\Omega) = \frac{\hat{p}_H(\Omega) \cdot s - \hat{p}_H^{(i)}(\Omega)}{s-1} \tag{3}$$

Now, $\hat{p}_H^{-(i)}(\Omega)$ is simply the selectivity that our estimator would have predicted if point $\vec{t}^{(i)}$ had not been part of the sample. Based on this, we can compute the adjusted estimation error $\mathcal{L}_{abs}\left(p\left(\Omega\right), \hat{p}_H^{-(i)}\left(\Omega\right)\right)$, which is the estimation error if $\vec{t}^{(i)}$ had been removed. The principle idea behind our maintenance algorithm is then simple: A sample point that significantly reduces the estimation error by its absence is likely misrepresenting the distribution in the data set and should be replaced. Accordingly, we define the *Karma* $K^{(i)}(\Omega)$ for sample point $\vec{t}^{(i)}$ as its impact on the estimation error:

$$K^{(i)}(\Omega) = s \cdot \Big(\mathcal{L}_{abs}(p\left(\Omega\right), \hat{p}_H(\Omega)) - \mathcal{L}_{abs}\Big(p\left(\Omega\right), \hat{p}_H^{-(i)}(\Omega)\Big)\Big) \tag{4}$$

We multiply by the sample size s to normalize the values to $[-1, 1]$: Karma values close to one correspond to sample points that significantly improved the estimation quality for query region Ω, while negative values are associated with points that had a negative impact. If the selectivity is overestimated, points contributing to the overestimation will be penalized, while points outside of the query region – or those without significant contributions – will be rewarded, vice-versa for underestimated selectivities.

By aggregating Karma values over a sequence of query regions $[\Omega_1, ..., \Omega_n]$, we obtain an indicator for the contribution of sample points over multiple queries. Accordingly, we recursively define our notion of *cumulative Karma* via the following recursion:

$$K^{(i)}([\Omega_1, ..., \Omega_n]) = \begin{cases} \alpha \cdot K^{(i)}(\Omega_n) + \\ \quad (1-\alpha) \cdot K^{(i)}([\Omega_1, ..., \Omega_{n-1}]) & n > 1 \\ \alpha \cdot K^{(i)}(\Omega_n) & n = 1 \end{cases} \tag{5}$$

In this equation, $\alpha \in (0, 1]$ is a constant factor for applying exponential smoothing to limit the influence of historic Karma values. This approach helps us to achieve faster reactivity on changing data, as well as to improve the method's robustness with respect to outliers.

The cumulative Karma is used as the foundation for our heuristic sample maintenance with focus on reestablishing good estimation results. This is done by resampling points with large negative Karma values as they are likely to misrepresent the true data distribution. This approach relieves us from mirroring all changes to the sample. $K^{(i)}$ can be computed easily on top of our KDE-based estimator: The calculation can be performed by executing one additional embarrassingly parallel computation on the GPU and allocating an additional field for the most recent values of $K^{(i)}$ for all points in the sample.

Note that this algorithm does not provide true sample uniformity, but instead aims at maintaining a sample that fits the underlying distribution in the queried regions.

5. DEMONSTRATION

In our demonstration, we first introduce our implementation in PostgreSQL and give an overview of the modifications that were applied. Afterwards we provide an interactive graphical evaluation of several sample maintenance algorithms under variable parameters and workload characteristics.

5.1 System Overview

All presented algorithms were integrated into the open-source database PostgreSQL 9.3.1. The GPU-accelerated algorithms were implemented in a hardware-oblivious way using OpenCL, which allows us to operate on all devices supporting the standard – including graphics cards, and multi-core CPUs [12]. We provide PostgreSQL control variables to control the KDE-based selectivity estimation for selected tables and to select the sample maintenance method. Besides integrating the use of KDEs in the estimator for qualifying queries, we added hooks after query executions, insertions and deletions, which are used to call the selected sample maintenance algorithms.

5.2 Compared Methods

We compare the following sample maintenance algorithms during our interactive presentation:

No maintenance (NONE) is our first baseline. In this method, we do not perform any sample maintenance, demonstrating the severity of estimation error degradation as updates are applied to the database.

Correlated Acceptance Rejection Sampling (CAR) is used as a baseline for existing maintenance algorithms and is implemented as explained in Section 2.

Periodic Random Replacement (PRR) is our third baseline. It replaces a random item from the sample with a newly sampled item every n changes to the base data.

Triggered Karma Replacement (TKR) replaces sample points when their cumulative Karma goes below a given threshold γ. A bitmap identifying points that will be resampled has to be calculated after Karma calculation and is transferred to the CPU to trigger resampling.

Periodic Karma Replacement (PKR) periodically replaces the sample point with the worst cumulative Karma every m queries. The sample point $\vec{t}^{(i)}$ with the minimum Karma is efficiently identified on the graphics card via a parallel reduction scheme [13].

5.3 Demonstration Overview

At the beginning of the demonstration, the user can choose from pre-selected dataset choices, each with different properties and sizes. We then collect and transfer a new sample for the selected dataset to the graphics. Afterwards, the user is prompted to select and configure a maintenance algorithm, and to specify characteristics of the query workload (e.g. the probability of insertions and deletions).

After starting the experiment, we continuously run random selection queries, plotting the average selectivity estimation error, as well as the number of transfered tuples that were required for the maintenance algorithms. This allows the user to inspect in real-time how the sample quality, and the required data transfers develop. The presentation will be delivered with an interface similar to Figure 2.

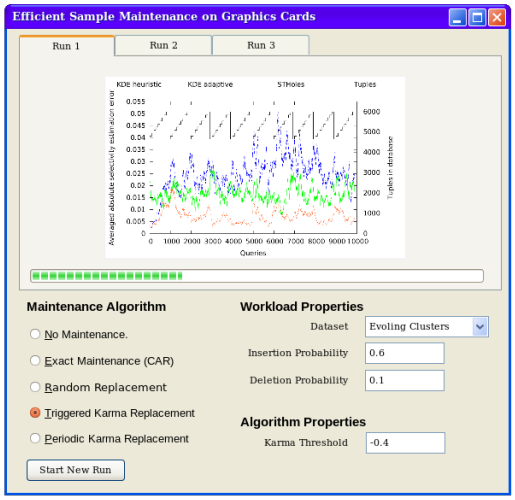


Figure 2: Overview of the demonstration interface: The user can select the desired sample maintenance method and specify algorithm and workload properties. When the user starts the configured experiment, we plot the average estimation-error from the sample, the required transfers across the PCI Express bus, and the cumulative updates to the database.

6. REFERENCES

[1] *Multivariate Density Estimation - Theory, Practice and Visualization*. John Wiley & Sons, Inc., 1992.

[2] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *ACM SIGMOD Record*, volume 28, pages 181–192. ACM, 1999.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[4] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[5] B. Blohsfeld, D. Korus, and B. Seeger. A comparison of selectivity estimators for range queries on metric attributes. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 239–250, New York, NY, USA, 1999. ACM.

[6] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[7] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data transfer matters for gpu computing. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 275–282, Dec 2013.

[8] 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*, pages 595–606. VLDB Endowment, 2006.

[9] P. B. Gibbons, Y. Matias, and V. Poosala. Maintaining a random sample of a relation in a database in the presence of updates to the relation, Jan. 4 2000. US Patent 6,012,064.

[10] D. Gunopulos, G. Kollios, J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, Apr. 2005.

[11] M. Heimel and V. Markl. A first step towards gpu-assisted query optimization. In *ADMS@ VLDB*, pages 33–44. Citeseer, 2012.

[12] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[13] D. Horn. Stream reduction operations for gpgpu applications. *Gpu gems*, 2:573–589, 2005.

[14] P.-A. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 175–186. ACM, 2007.

[15] R. J. Lipton, J. F. Naughton, and D. A. Schneider. *Practical selectivity estimation through adaptive sampling*, volume 19. ACM, 1990.

[16] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *Data Engineering, 1992. Proceedings. Eighth International Conference on*, pages 632–641, Feb 1992.

[17] H. Toivonen et al. Sampling large databases for association rules. In *VLDB*, volume 96, pages 134–145, 1996.

[18] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.