Letting the application drive the sample maintenance
EDBT 2015
1. What are Kernel Density Estimators (KDEs)?
2. KDEs for selectivity estimation
3. KDE evaluation on GPUs
4. Sample Maintenance on GPUs
5. Traditional Sample Maintenance: CAR
6. The Karma Score
7. Experiments
What are Kernel Density Estimators (KDEs)?

- KDEs estimate probability density functions
  - Based on a sample

- They can be used for selectivity estimation
  - KDEs deliver high quality selectivity estimates
  - KDEs can be efficiently evaluated on GPUs
  - More details soon to be published

KDEs for selectivity estimation (1)

- Step 1: Draw a uniform sample $S = \{\vec{t}^{(1)}, \vec{t}^{(2)}, \ldots, \vec{t}^{(s)}\} \subseteq R$
- Step 2: Center so called Kernels $K_H$ around sample points

\[
K_H(\vec{t}^{(i)} - \vec{x})
\]
Step 3: The probability density function estimate is given by averaging over these points

\[ \hat{p}_H(\vec{x}) = \frac{1}{s} \sum_{i=1}^{s} K_H(\vec{t}^{(i)} - \vec{x}) \]

Step 4: Selectivity estimates for a region \( \Omega \) are given by integrating the function

\[ \hat{p}_H(\Omega) = \int_{\Omega} \hat{p}_H(\vec{x}) d\vec{x} = \frac{1}{s} \sum_{i=1}^{s} \int_{\Omega} K_H(\vec{t}^{(i)} - \vec{x}) \hat{p}_H^{(i)}(\Omega) \]
KDE evaluation on GPUs

- Scanning large samples is rather inefficient
- That's what we have GPUs for!
If the data stays constant, this is it:

However, if we are subject to Insertions/Deletions/Updates…
- We would not want to create a new sample
- We need sample maintenance

However, bus bandwidth is limited
- We want sample maintenance to be transfer-efficient
- Good estimation quality, economical bandwidth usage
Correlated Acceptance Rejection (CAR)

**on_insertion**\((\text{inserted}_t)\){
  \[
  r = \text{draw from BINOM}(|S|, 1/|R|) \\
  \text{remove } r \text{ random elements from } S \\
  \text{fill } S \text{ with instances of } \text{inserted}_t
  \]
}

**on_update**\((\text{before}_t, \text{after}_t)\){
  \[
  \text{replace } \text{before}_t \text{ in } S \text{ with } \text{after}_t
  \]
}

**on_deletion**\((\text{deleted}_t)\){
  \[
  \text{remove instances of } \text{deleted}_t \text{ in } S \\
  \text{fill } S \text{ with sampled tuples from } S
  \]
}

**Insertions only:**
- We only transfer inserted\(_t\) if necessary
- Probability decreases with growing \(R\)

**For every update, we need to transfer:**
- before\(_\text{tuple}\)
- after\(_\text{tuple}\)

**For every deletion, we need to transfer:**
- deleted\(_\text{tuple}\)
- Map identifying the deleted tuples
- Sampled tuples

Can we be more transfer-efficient for change-intensive workloads by dropping uniformity?
Key facts:
- Every sample point contributes probability mass independent of the others
- We know the true selectivity for a query after execution $p(\Omega)$
- How does removing a point influence the estimate?
  - Degrades the estimate: Good point
  - Improves the estimate: Bad point

\[
\hat{p}_H^{-\langle i \rangle}(\Omega) = \frac{\hat{p}_H(\Omega) \cdot s - \hat{p}_H^{\langle i \rangle}(\Omega)}{s - 1}
\]
The Karma Score

- The change on the estimation error gives us the Karma score:
  \[ K^{(i)}(\Omega) = |S| \cdot \left( \mathcal{L}_{abs}(p(\Omega), \hat{p}_H(\Omega)) - \mathcal{L}_{abs}(p(\Omega), \hat{p}_H^{-}(\Omega)) \right) \]
  - Beneficial tuple: Positive score
  - Harmful tuple: Negative score

- We aggregate the Karma value over incoming queries
  \[ K^{(i)}_{t+1} = \min \left( K^{(i)}_t + K^{(i)}(\Omega), K_{\text{max}} \right) \]
  - Can be computed in parallel for every sample point

- On the GPU:
  - One additional array on device memory
  - One parallel computation
on_query(estimated_sel, true_sel) {
    update_karma(estimated_sel, true_sel)
    resample point with karma worse than threshold
}

- For every query, we have to transfer:
  - A map identifying sampled tuples exceeding the threshold
  - Sampled tuples
Experiments

- Relatively new laptop (Dell Precision M4800)
  - Ubuntu Linux
  - 32 GB of RAM
  - Core i7 i7-4810MQ
  - Nvidia Quadro K2100M

- Moving clusters workload
  - Clusters are inserted along a line
  - Insertions and queries are occurring alternatingly
  - Periodically, old clusters are removed
  - Cluster size: 3000 tuples
  - Queries: 10000 queries
  - 10 clusters inserted
  - 10 clusters removed

- Error averaged over the last 100 queries
- Transfers are measured in μ seconds
No sample maintenance is no solution

- No sample maintenance
The Karma Score works in terms of error

- Resampling a random tuple from the sample every query (PRR)
- Replace point with worst Karma
Triggered Karma Replacement works in terms of error

- **CAR**
- **TKR (Threshold: -2, Limit: 4)**
TKR is transfer-efficient for the workload

- CAR
- TKR (Threshold: -2, Limit 4)
- None
The lower the threshold, the better the estimation

- TKR (Threshold: -1 | -2 | -4 | -8, Limit: 4)
The lower the threshold, the more transfers are necessary.

- TKR (Threshold: -1 | -2 | -4 | -8, Limit: 4)
Having the $K_{\text{max}}$ cap is important for fast convergence

$$K_{t+1}^{(i)} = \min\left(K_t^{(i)} + K^{(i)}(\Omega), K_{\text{max}}\right)$$

TKR (Threshold: -2, Limit: None)
TKR (Threshold: -2, Limit: 4)
Increasing the number of queries, increases the number of transfers

- Increase number of queries to 30000

- CAR
- TKR (Threshold: -2, Limit 4)
CAR

- Hyper-parameter free
- Transfer-efficient insertion-only workloads
- Transfer-efficient for $|\text{Changes}| << |\text{Queries}|$
- Guarantees a uniform sample

- Not transfer-efficient for $|\text{Changes}| >> |\text{Queries}|$

TKR

- Transfer-efficient for $|\text{Changes}| >> |\text{Queries}|$
- Can outperform CAR in terms of error

- Not transfer-efficient for $|\text{Changes}| << |\text{Queries}|$
- Contains hyper-parameters
- Does not guarantee statistical properties