Error-driven sample maintenance in GPU-accelerated Kernel Density Estimation

Letting the application drive the sample maintenance
EDBT 2015

Fachgebiet Datenbanksysteme und Informationsmanagement
Technische Universität Berlin

http://www.dima.tu-berlin.de/

Agenda

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)?

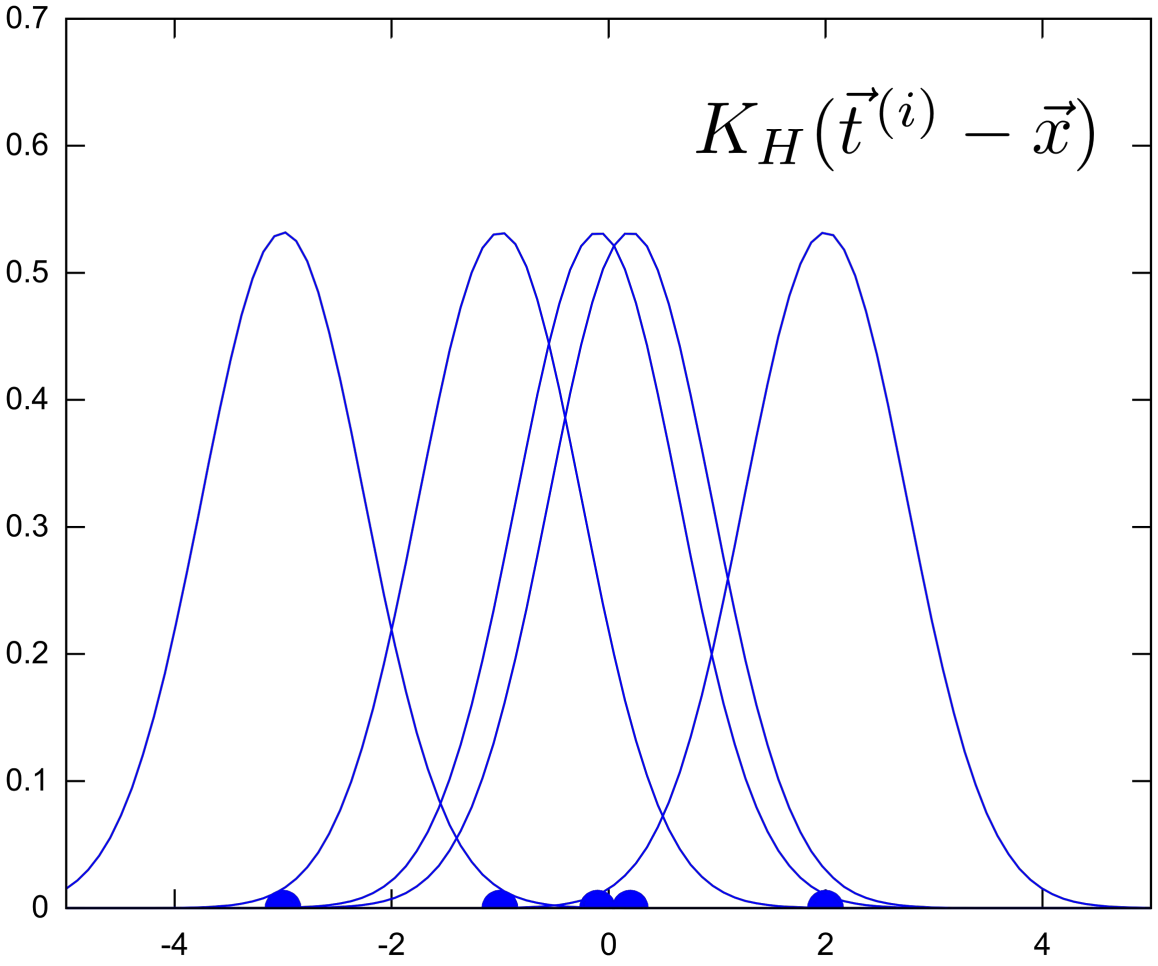
- KDEs estimate probability density functions
  - Based on a sample

- 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

Heimel, Kiefer, Markl - Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. SIGMOD 2015

KDEs for selectivity estimation (1)

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

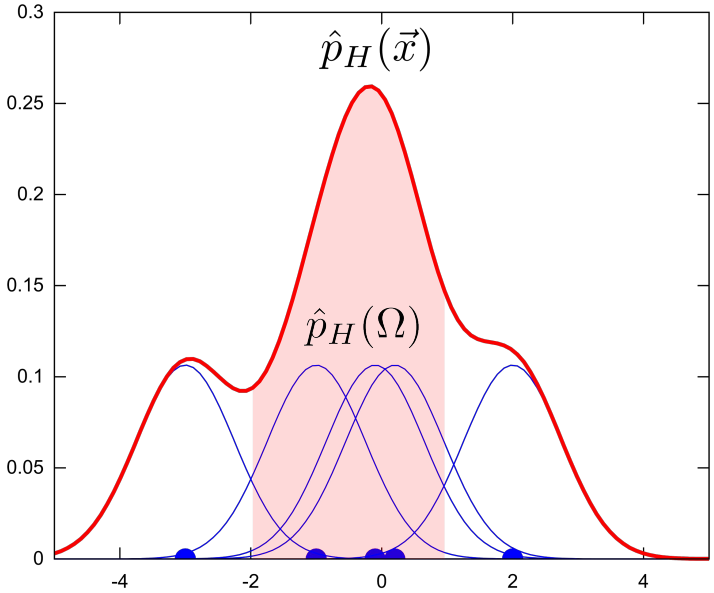


- Step 3: The probability density function estimate is given by averaging over these points

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

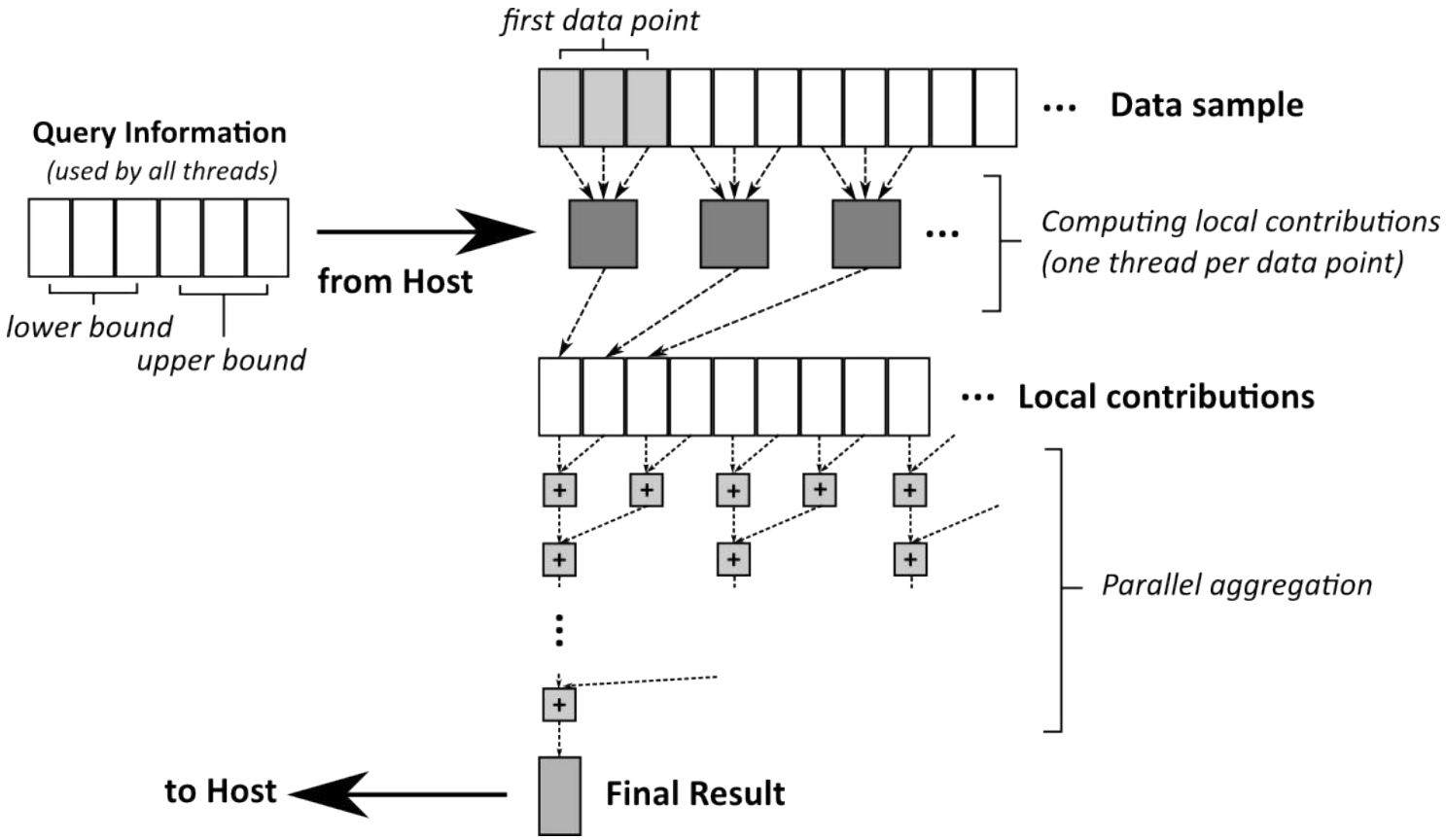
- Step 4: Selectivity estimates for a region Ω are given by integrating the function

$$\hat{p}_H(\Omega) = \int_\Omega \hat{p}_H(\vec{x})d\vec{x} = \frac{1}{s}\sum_{i=1}^{s} \underbrace{\int_\Omega K_H(\vec{t}^{(i)} - \vec{x})}_{\hat{p}_H^{(i)}(\Omega)}$$

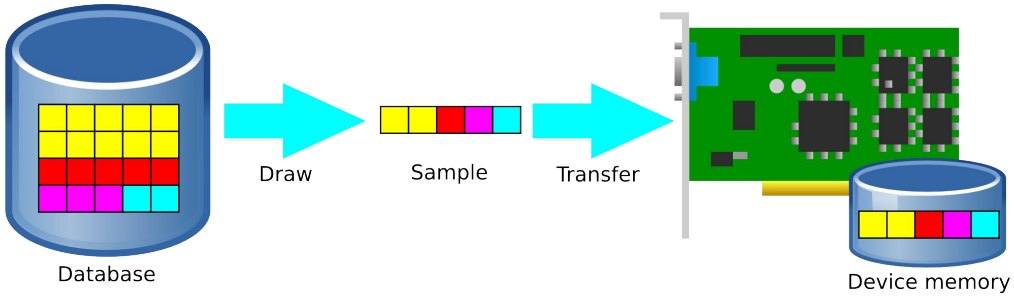


KDE evaluation on GPUs

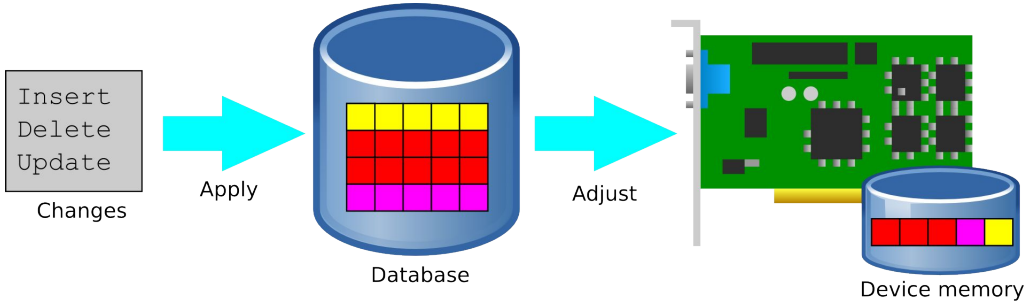
- Scanning large samples is rather ineffcient
- That's what we have GPUs for!



- 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 bandwdith usage**

Traditional Sample Maintenance: Correlated Acceptance Rejection (CAR)

Correlated Acceptance Rejection (CAR)

```
on_insertion(inserted_t){
    r = draw from BINOM(|S|, 1/|R|)
    remove r random elements from S
    fill S with instances of inserted_t
}

on_update(before_t, after_t){
    replace before_t in S with after_t
}

on_deletion(deleted_t){
    remove instances of deleted_t in S
    fill S 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_tuple
- after_tuple

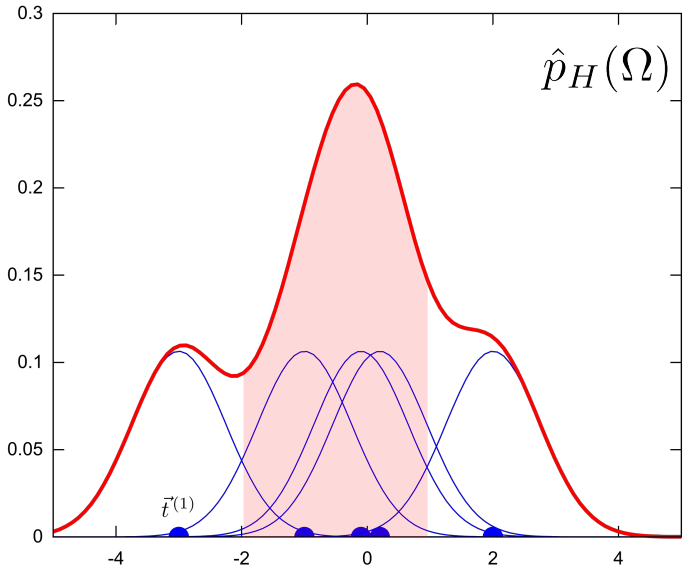
! For every deletion, we need to transfer:

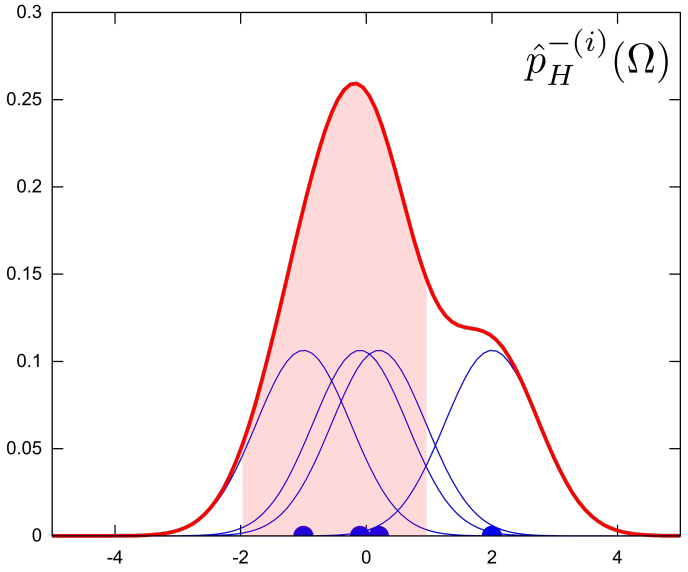
- deleted_tuple
- Map identifying the deleted tuples
- Sampled tuples

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

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





- The change on the estimation error gives us the Karma score:

$$K^{(i)}(\Omega) = |S| \cdot \left(\mathcal{L}_{abs}(p(\Omega), \hat{p}_H(\Omega)) - \mathcal{L}_{abs}\left(p(\Omega), \hat{p}_H^{-(i)}(\Omega)\right)\right)$$

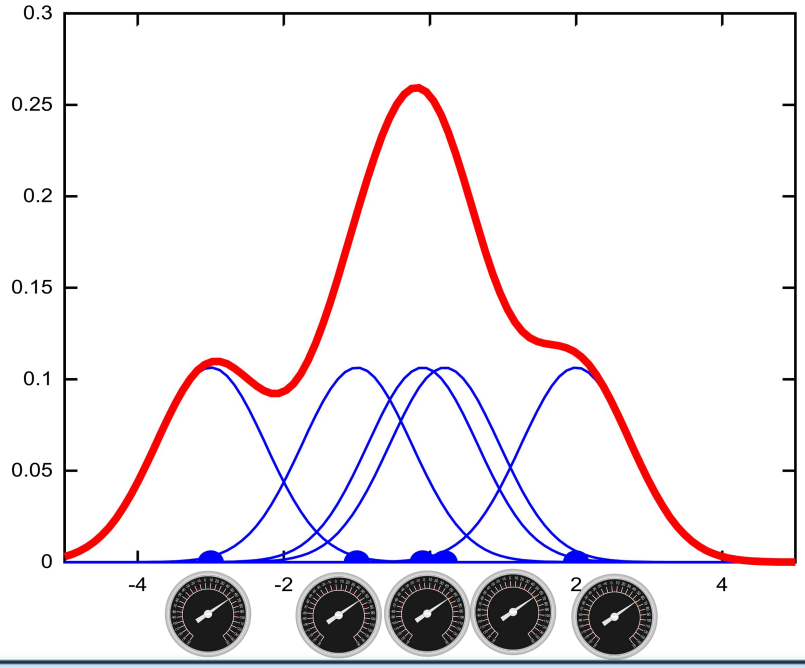
  - Benefitial tuple: Positive score
  - Harmful tuple: Negative score

- We aggregate the Karma value over invoming queries

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

- Can be computed in parallel for every sample point

- On the GPU:
  - One additional array on device memory
  - One parallel computation



The Karma Score: Triggered Karma Replacement

```
on_query(estimated_sel, true_sel){
    update_karma(estimated_sel, true_sel)
    resample point with karma worse than threshold
}
```

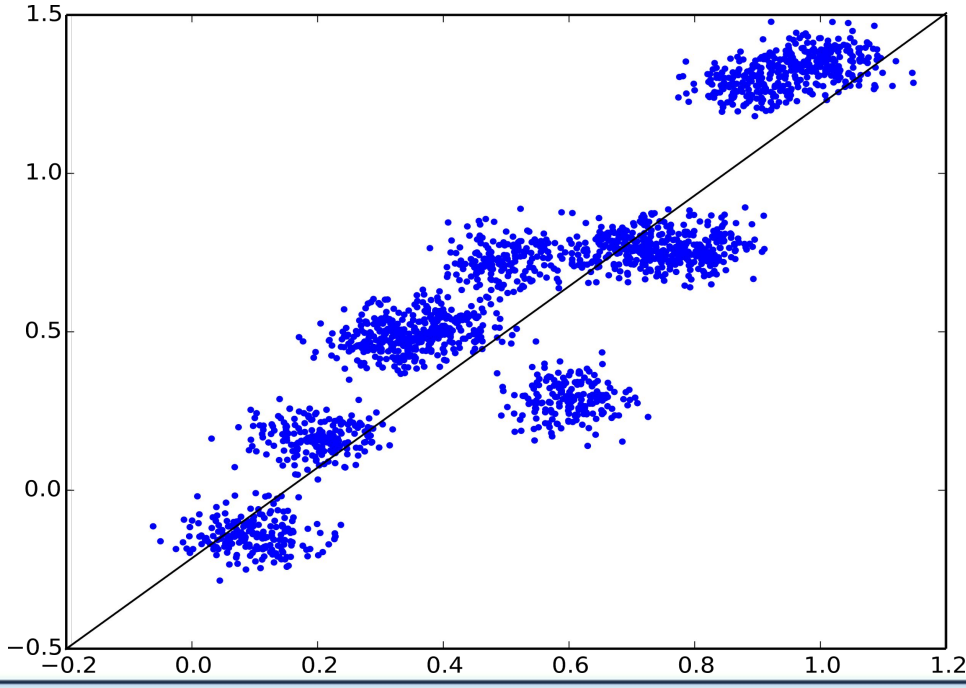
- For every query, we have to transfer:
  - A map identifying sampled tuples exceeding the trehshold
  - Sampled tuples

Experiments

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

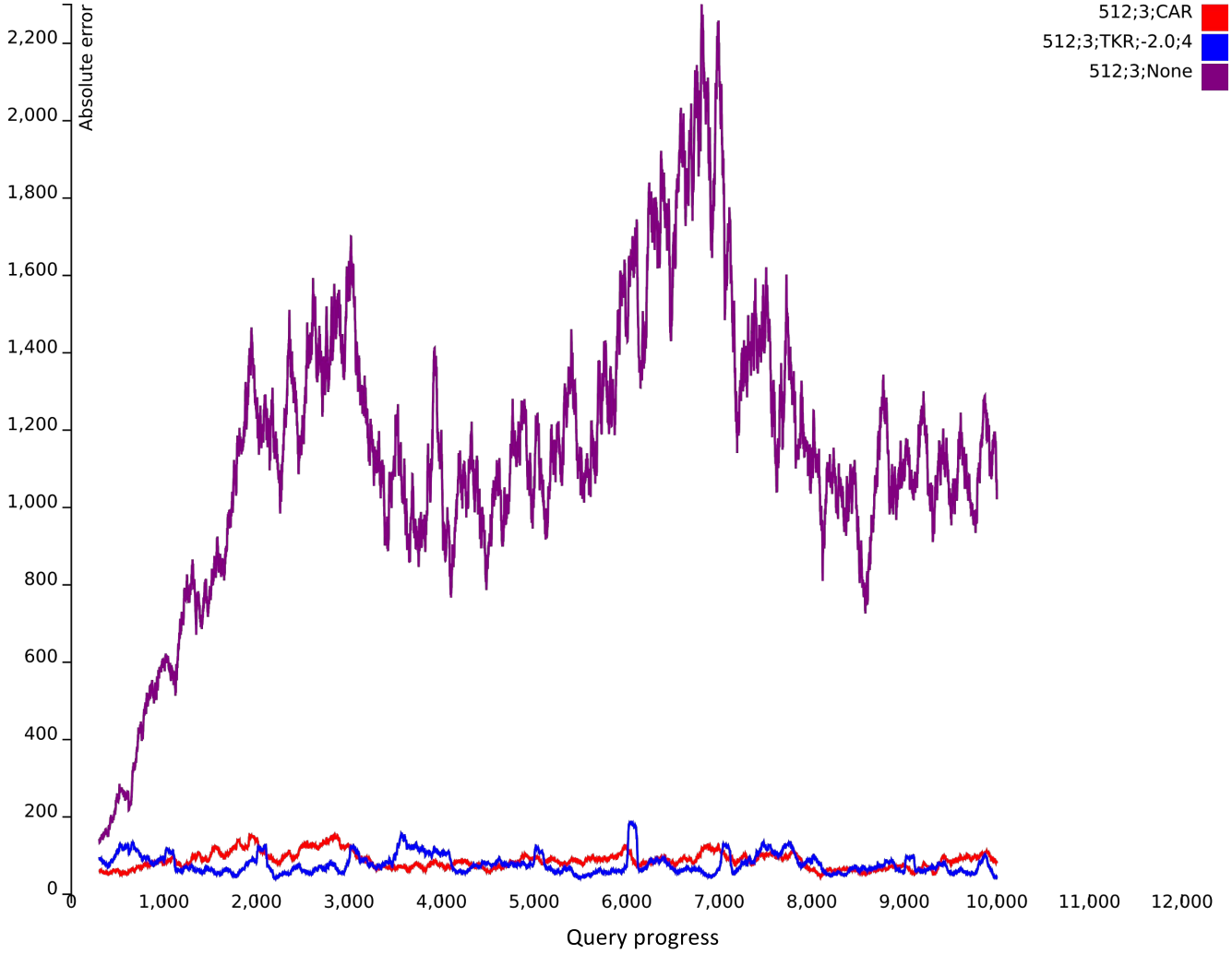
- Moving clusters workload
  - Clusters are inserted along a line
  - Insertions and queries are occuring alternatingly
  - Periodically, old clusters are removed
  - Cluster size: 3000 tuples
  - Queries: 10000 queries
  - 10 clusters inserted
  - 10 clusters removed

- Error averaged over the last 100 queries
- Transfers are measured in µ seconds



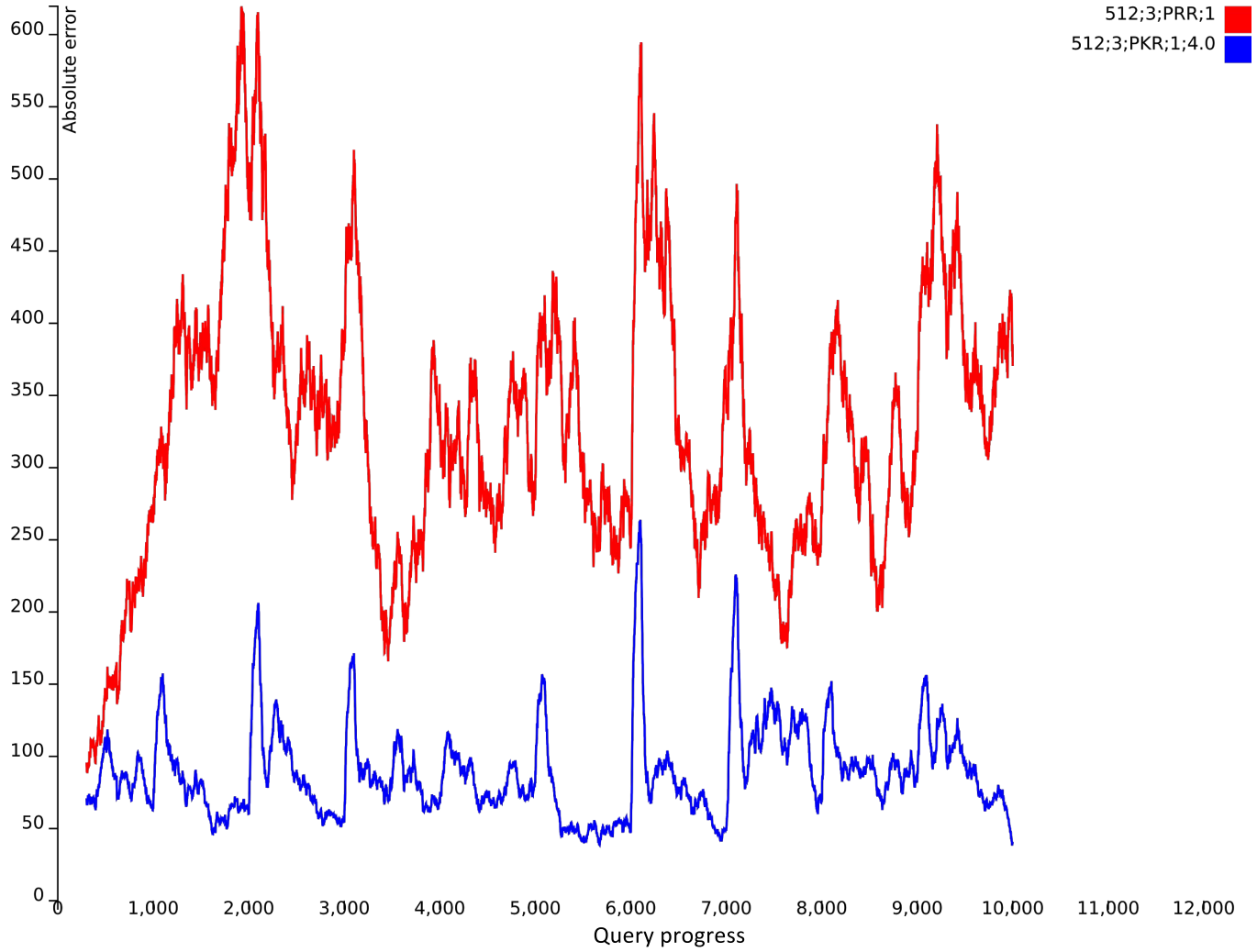
No sample maintenance is no solution

- No sample maintenance



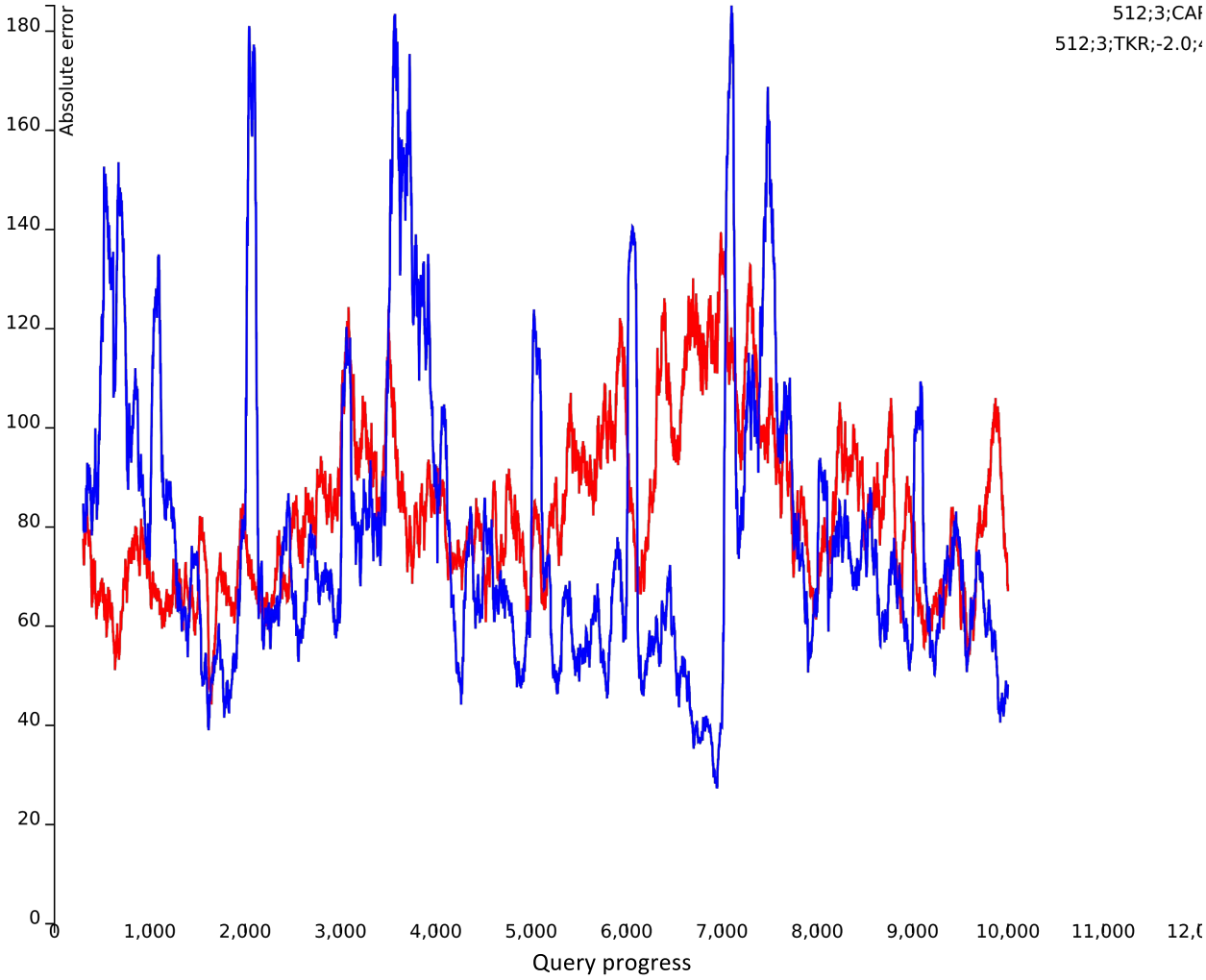
The Karma Score works in terms of error

- Resampling a random tuple from the sample every query (PRR)
- Replace point with worst Karma



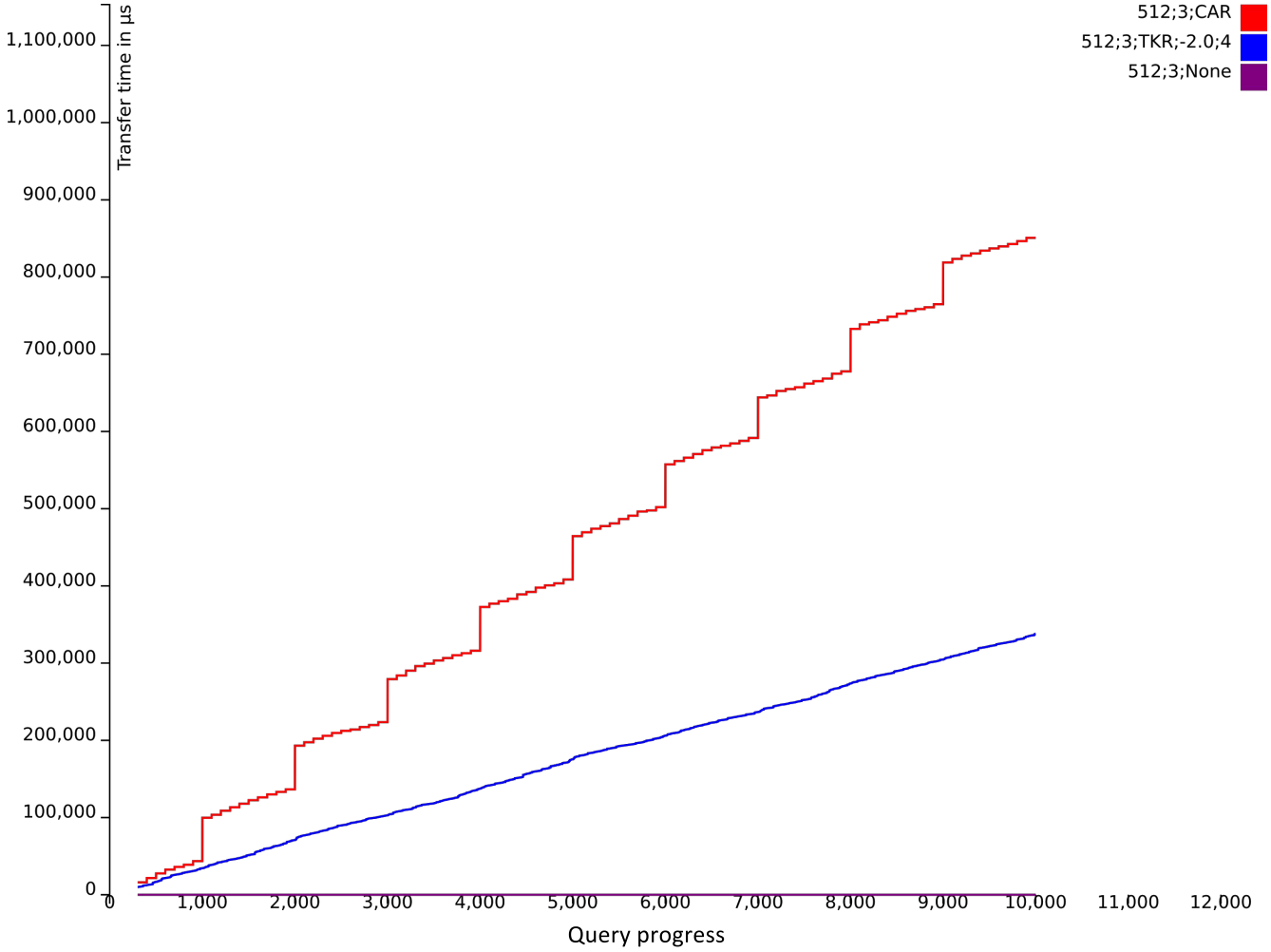
Triggered Karma Replacement works in terms of error

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



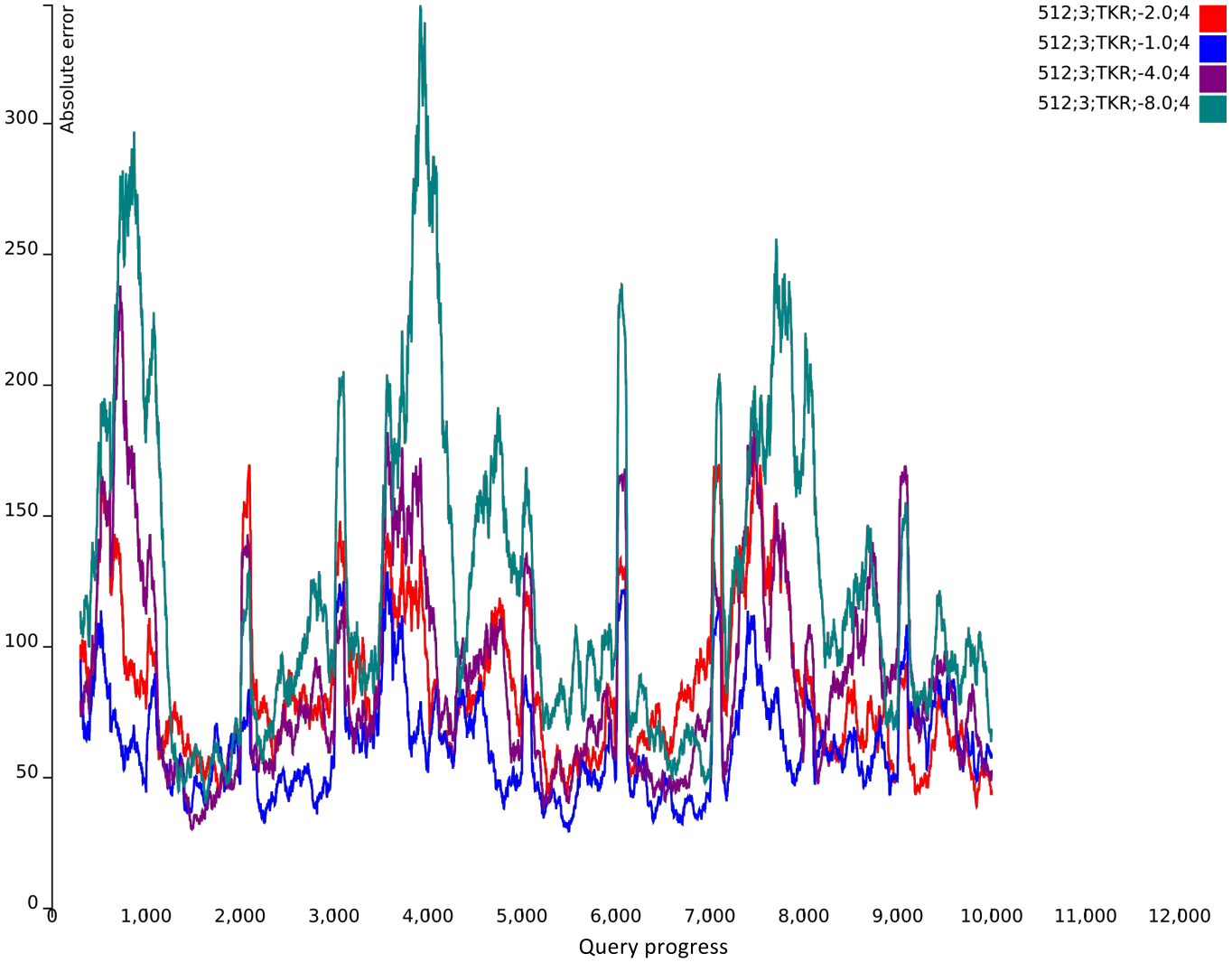
TKR is transfer-efficient for the workload

- CAR
- TKR (Threshold: -2, Limit 4)
- None



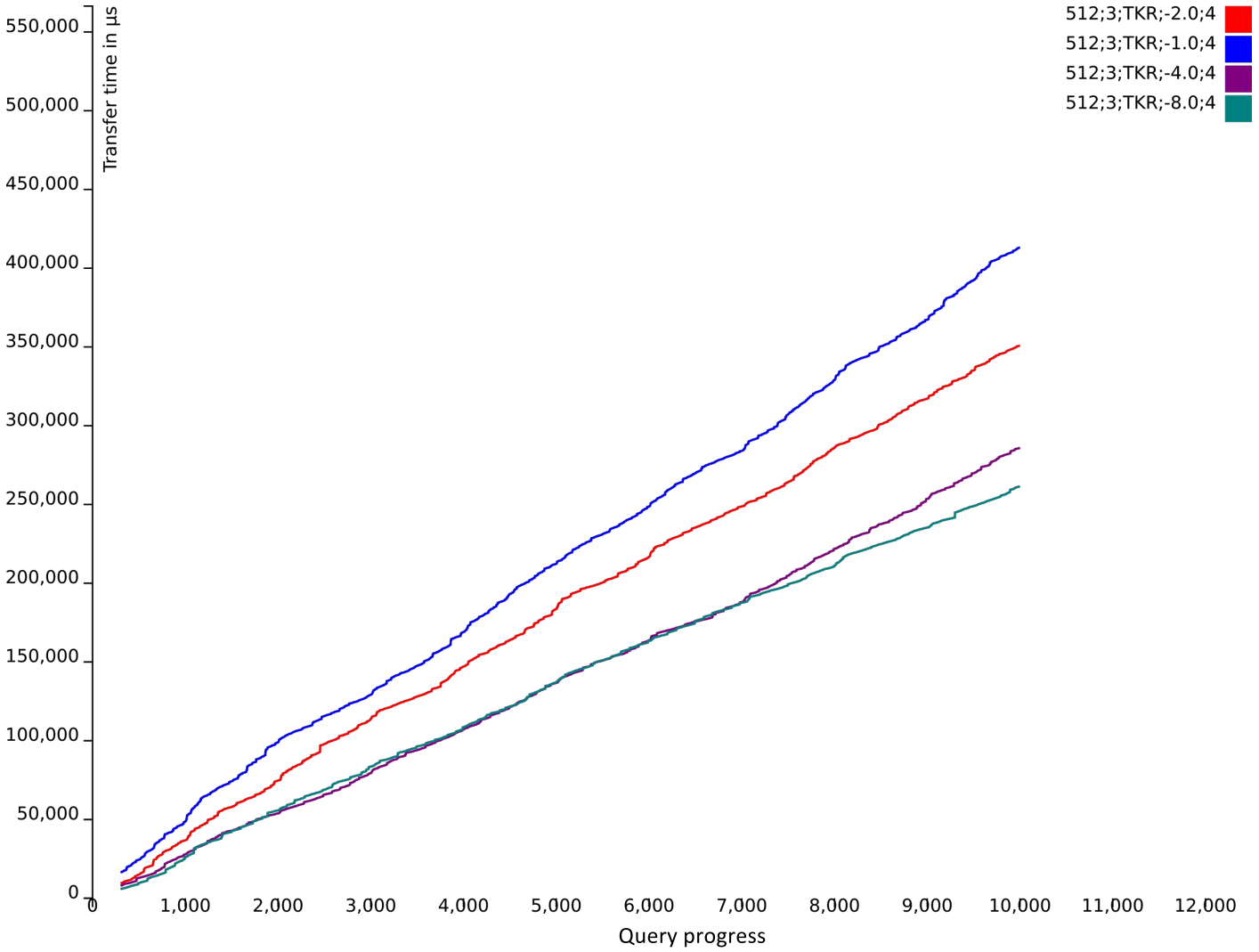
The lower the threshold, the better the estimation

- TKR (Threshold: -1 | -2 | -4 | -8, Limit: 4)



The lower the threshold, the more transfers are necessary

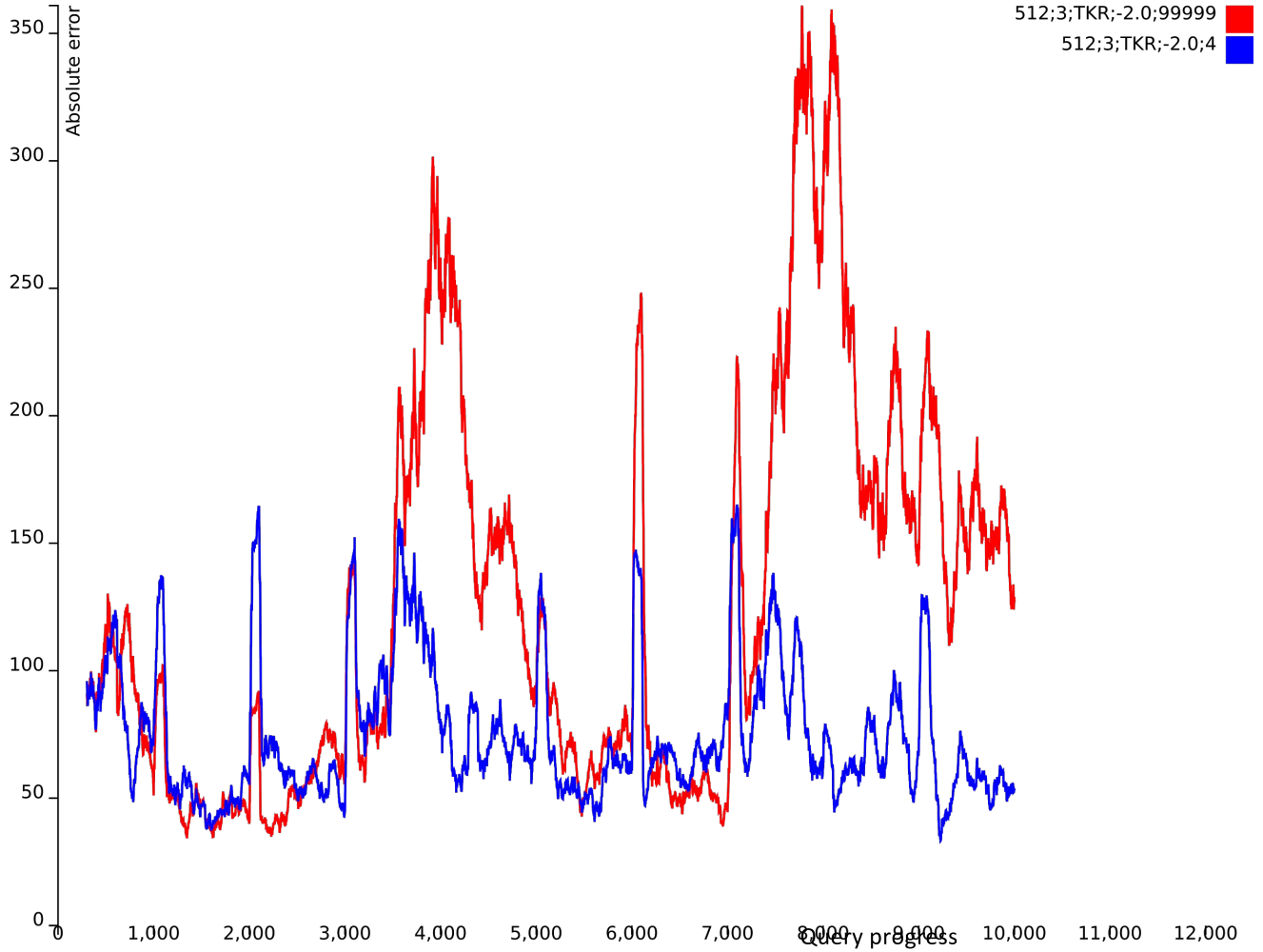
- TKR (Threshold: -1 | -2 | -4 | -8, Limit: 4)



Having the $K_{\max}$ cap is important for fast convergence

- TKR (Threshold: -2, Limit: None)
- TKR (Threshold: -2, Limit: 4)

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

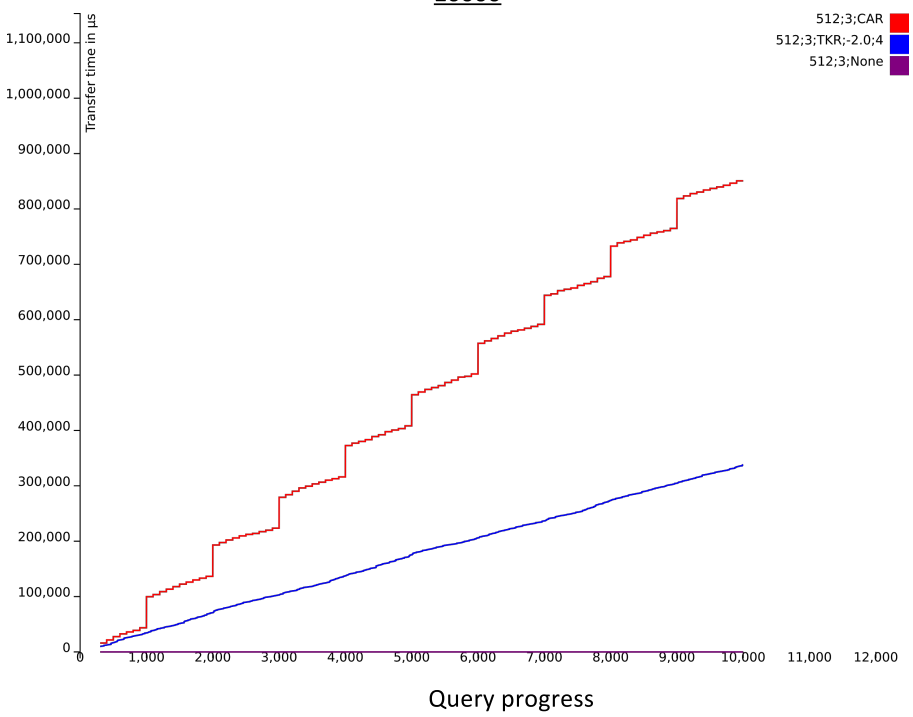


Increasing the number of queries, increases the number of transfers

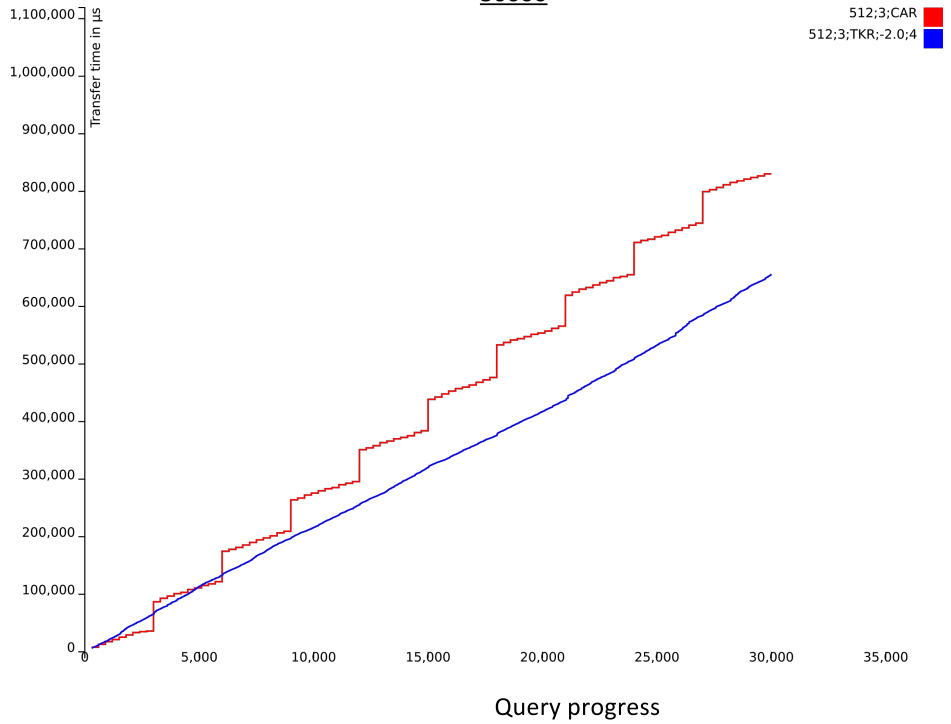
- Increase number of queries to 30000

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

10000



30000



Conclusion

- **CAR**

- Hyper-parameter free
- Transfer-efficient insertion-only workloads
- Transfer-efficient for
  |Changes| << |Queries|
- Guarantees a uniform sample

- Not transfer-efficient for
  |Changes| >> |Queries|

- **TKR**

- Transfer-efficient for
  |Changes| >> |Queries|
- Can outperform CAR in terms of error

- Not transfer-efficient for
  |Changes| << |Queries|
- Contains hyper-parameters
- Does not guarantee statistical properties